WEEK 1 – HISTORY (& ASSEMBLY, I MOVED IT AFTER ISA)

HISTORY

Architecture = interconnection of components

Hardware is interconnected via "buses" (circuits)

Software is interconnected via "interfaces" (api)

PRE-HISTORY: CALCULATORS AND PROGRAMMABLE MACHINES (1700-1930)

Calculators:

Machines of **Pascal** and **Leibniz** were mechanical devices

- No memory or program
- Leibniz used binary system (1705)
- A single operation at a time
- Only simple operations (+,-,*,/)

Programmable machines:

- Mechanical **Music** Instruments
  - Bagdad 9th Century
  - "Carillons"
- **Chess**
  - Mechanical Turk (1770) (fraud, man inside a box)
- **Weaving** machines
  - **Jacquard Loom** (Head) (1801)
  - Used **Punch cards**

Difference (polynomials) engine

- Invented by **Johann Helfrich von Muller** 1786
- Extended by **Charles Babbage** (1822) but never finished

Analytical Engine (First Conceptual CPU)

- Designed by **Charles Babbage** (inspired by Jacquard punch cards and von Muller calculator)
- Never completed, but brought the **Instruction Set Architecture** concept of a **CPU**:
- **Arithmetic Unit + IFs + Memory** (with stored-program and variables) **+ I/O devices**.
  - Program contained calculations + order sequences

First computer programmer (for non-finished Analytical Engine): Ada countess of Lovelace 1840s

- **First computer algorithm: Note G**, in Assembly
- Math algorithm to generate Bernoulli Numbers

Analog Computers: Vannevar Bush 1931.

- **First** systems that enabled significant **reduction** of calculation time
- Used nomograms and slide rules. **Graphical tools** designed to allow the approximate graphical computation of a mathematical function/operation.

1ST GENERATION: ELECTRO-MECHANICAL (1930-1950)

Boosted by World War 2 (1939-1945).

Electro-mechanical devices:

ASCC: Automatic Sequence Controlled Calculator

- Built by **Howard Aiken** 1937-1944
- **First general purpose** digital computer
- 750,000 components
- 5 tons
- 100 times faster in theory, 3-5 times faster in practice (component failures)

ENIAC: Electronic Numerical Integrator and Computer

- Built by **John Mauchly** and **John Presper Eckert** 1943-47
- **First all-electronic computer**
- But **international patent** won later by **John Atanasoff** in 1973 (computer already in 2nd generation).
- 18k tubes of 5-10cm
- 150 kW dissipation
- 30 tons
- 1000 bits of memory
- 20 hours to 20 seconds.
- 10 tubes broke on power up
- Difficult to program
- Not very flexible
- Technologically complex
- Small memory
- Literal **bugs** (and origin of software bug term) would break it

EDVAC: Electronic Discrete Variable Automatic Computer

- Built by **Mauchly** and **Eckert** 1948-49
- Basis of **Von Neumann Architecture**
- Mean Time to Failure (MTTF) 8 hours

2ND GENERATION: TRANSISTORS (1955-1975)

Transistors:

- Reliable
- Less power
- **U**. **Manchester** world's **first transistorized computer** 1953
- **Bell Labs** 1948
- **DEC PDP-1** 1959
  - **Hacker culture**
  - **First game (Spacewar)**
- **1st Supercomputer: Cray's CDC 6600 – 10MFLOPS** floating point operations per second

3RD GENERATION: MICROPROCESSORS (1960-TODAY)

Integrated Circuits:

Enabled small low-cost microprocessors

1st CPU: Intel 4004 * 108KHz

Apple 1978 **first B2B computer**

IBM 1980 **Personal Computer first commercial computer**

- Blueprint for today's PCs
- Revolutionized the market
- Open standards and friendliness to third-party hardware and software developers

PERIPHERALS (I/O DEVICE) - BOTH 2ND AND 3RD GENERATION

First monitor 1951 was US army's display system

First mouse 1968 **Doug Engelbart** "X-Y Position Indicator for a Display System"

4TH GENERATION: MULTI-COMPUTING (1969-TODAY)

Roots of the Internet:

ARPANET 1965-1969

- **Leonard Kleinrock** develops queuing Theory
- 4 computers at UC Santa Barbara, UC Los Angeles, Stanford, U Utah

1972 ARPANET public + Email

TCP/IP at Stanford 1974 (universal protocols between different machine systems)

1982 ARPANET +TCP/IP = early Internet

Cloud computing (Server farms, multi-core)

File/video/… sharing; IoT; Social Media

WEEK 2 – LOGIC CIRCUITS

John Vincent Atanasoff Intermezzo. Inventor of Digital Computer 1930s. Programmable devices that compute arbitrary arithmetic or logical operations, being able to perform more than one function. Use digital rather than analog components.

Atanasoff's principles of digital computers:

1. Use **binary** information bits
2. Use **electricity** and electronics instead of mechanical devices
3. **Memory** based on **capacitors**
4. Computation by **Boolean algebra**

Unit of Information:

Computers consist of digital (binary circuits)

bit (**b**inary dig**it**) 0 = off, 1 = on

Two interpretations of bits:

- **Arithmetic**: as **data** values
- **Logic**: as **truth** values (false or true)

Bit Strings: Groups of bits, which can be given a ***specific*** meaning

BOOLEAN ALGEBRA: USES 2 VALUES

A computer can transform Bit Strings (expressions) into other strings (results). 1 + 2 = 3 -> 01 XOR 10 = 11

George Boole 1854 created this algebra that can compute regular arithmetic.

Commutative law: x+y = y+x, x*y=y*x (order doesn't matter)

Distributive Law: x(y+z) = xy+xz

Associative law: (x+y)+z = x+(y+z). (no bracket) Mult is also associative. So AND and XOR are associative.

Different than school algebra: x+x = x, x*x = x (no squares, no 2)

Because an expression repeating the same (boolean 1/0) expression n times is redundant and its truth value remains the same.

Complement: the 1-x of something, in Boolean algebra = -x (not x)

Such that: x and not x = 0 (x(1-x) = x-x^2 = x-x = 0)

x or not x = 1 (x+(1-x) = 1-0 = 1). (1-x) = NOT x = $\overline{x}$ = ~x

Truth tables and computing functions

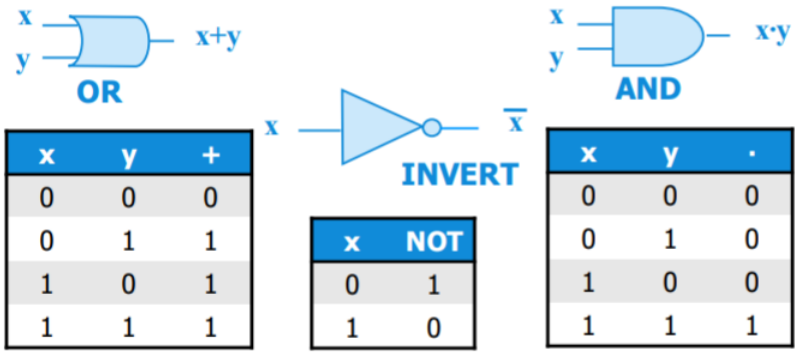
Sum of products form…

x	y	f(x,y) = x XOR y	f(x,y) = ~x*y + x*~y
0	0	0	***Minterm for f(…)=0 –Don't include--***
0	1	**1**	**Minterm for f(…)= 1 ~x*y**
1	0	**1**	**Minterm for f(…)= x*~y**
1	1	0	***only when listing Minterm with f(…)=1***

Any polynomial function can be constructed using Boolean algebra.

Any function has a Sum of Products form.

LOGIC GATES



x	y	+
0	0	0
0	1	1
1	0	1
1	1	1

x	NOT
0	1
1	0

x	y	·
0	0	0
0	1	0
1	0	0
1	1	1

XOR

NAND (not both)

NOR (neither)

x	y	+
0	0	0
0	1	1
1	0	1
1	1	0

x	y	NAND
0	0	1
0	1	1
1	0	1
1	1	0

x	y	NOR
0	0	1
0	1	0
1	0	0
1	1	0

De Morgan's Law: ~(xy) = ~x+~y (negation sign "flips" + for *)
~(x+y) = ~x*~y

KARNAUGH MAPS

Optimizing Sum-of-Products

Construction
- Rows/cols differ in 1 bit
- One cell per minterm
- Cell = 1 from truth table

Optimization
- Group all 1s
- Adjacent 1s, horiz./vert.
- Group size is power-of-2
- Largest groups

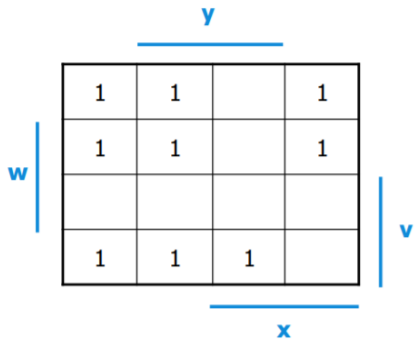
A different drawing
- Only represent areas where input = 1
- Labels where variables = 1

wrap around

vw \ xy	00	01	11	10
00	1	1	0	1
01	1	1	0	1
11	0	0	0	0
10	1	1	1	0

$f = \bar{v}\cdot\bar{x} + \bar{w}\cdot\bar{x} + v\cdot\bar{w}\cdot y + \bar{v}\cdot\bar{y}$

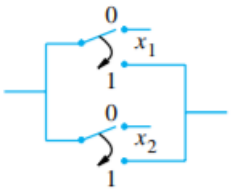
overlapping



$f = \bar{v}\cdot\bar{x} + \bar{w}\cdot\bar{x} + v\cdot\bar{w}\cdot y + \bar{v}\cdot\bar{y}$

Don't cares (marked as d/x/?) are jokers, can be used as 1 or 0 according to our needs. Used for minimization and for grouping larger more cells.

LOGIC GATES CIRCUITS

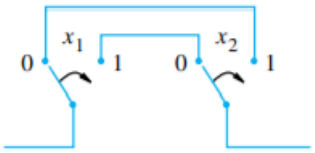


x_1	x_2	$f(x_1, x_2) = x_1 + x_2$
0	0	0
0	1	1
1	0	1
1	1	1

(b) Parallel connection (OR control)

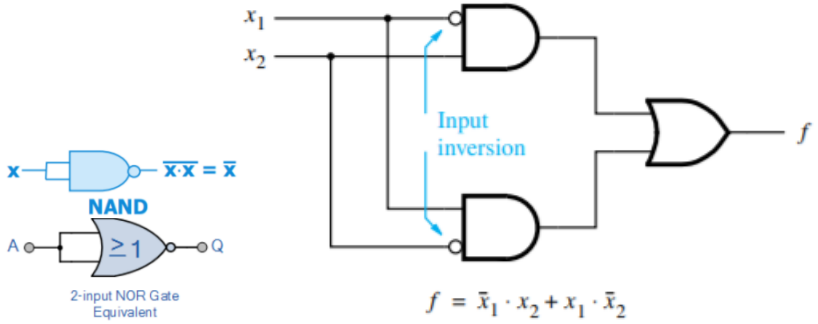
x_1	x_2	$f(x_1, x_2) = x_1 \cdot x_2$
0	0	0
0	1	0
1	0	0
1	1	1

(c) Series connection (AND control)



x_1	x_2	$f(x_1, x_2) = x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0

(d) EXCLUSIVE-OR connection (XOR control)



$f = \bar{x}_1 \cdot x_2 + x_1 \cdot \bar{x}_2$

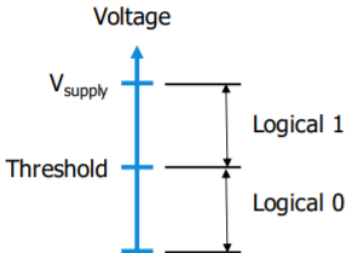
(a) Network for the XOR function

ELECTRONICS OF LOGIC GATES

Switching voltages:

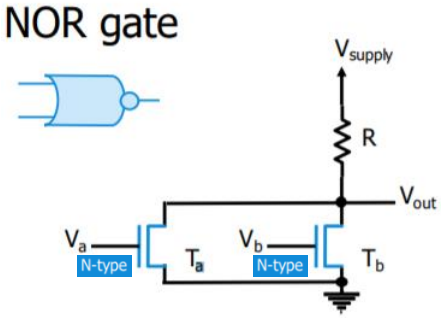
On = Voltage wants to go to **supply/Vout**(continuation) at **5V** (Vsupply)

Off = Voltage wants to go to **ground** and/or at **0V** (Vground) ⏚



If electrons go to ground with voltage that is a bad circuit because you are wasting energy **If both** a circuit connected back to **supply** /battery/Vout **and** a circuit connected to **ground**/source/earth are available in parallel **electrons** will **go** to **ground** (**Vout = 0**) because there are more positive charge electrons on earth than on supply, therefore attracting the flowing negative charged electrons and "draining" Vout from all the Voltage. If electrons go back to voltage without resistance too much energy will travel in the circuits causing a short circuit.

There is a Voltage **threshold** where the Voltage is **neither** logical **1 nor 0**.



NMOS TRANSISTOR GATE

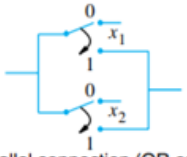
if +V -> closed gate (electron flow)

if 0V -> open gate (no flow)

Used for positive variables.

PULLDOWN: Conditions for Vout = 0

Va OR Vb: parallel and **N-type**



(b) Parallel connection (OR control)

PMOS TRANSISTOR GATE

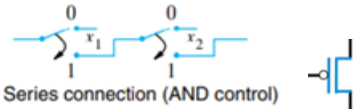
if +V -> open gate (no flow); **if 0V -> closed gate** (electron flow)

Used for negative variables.

PULLUP: Conditions for Vout = 1

x	y	NOR
0	0	1
0	1	0
1	0	0
1	1	0

~Va AND ~Vb. It would require a series connection and a P-type transistor (it conducts when 0V).



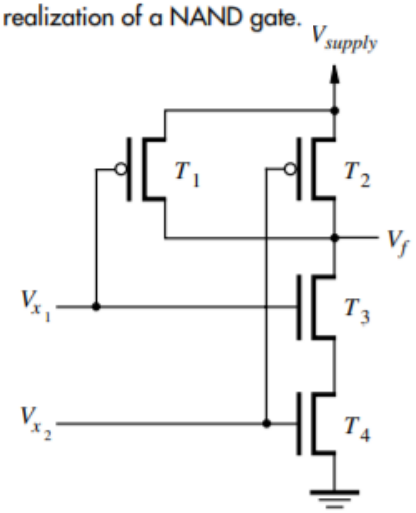
(c) Series connection (AND control)

Inverter gate: Quick read: If ~V connect to supply (and not ground), if V connect to ground (and not to supply).

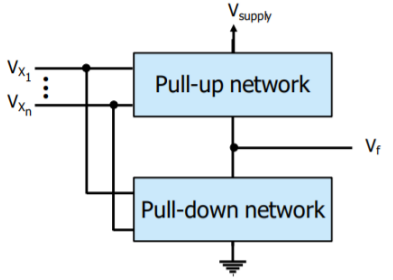
CMOS CIRCUIT:

Complementary metal-oxide semiconductor circuit. Combines PMOS and NMOS to avoid power consumption when connecting to ground. You can observe that the **Pull up network is exactly the opposite of the pulldown network using de Morgan's Law.**

CMOS realization of a NAND gate.



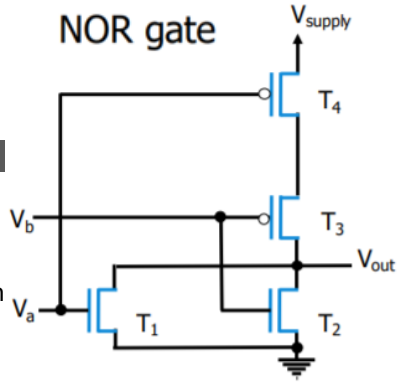
(a) Circuit



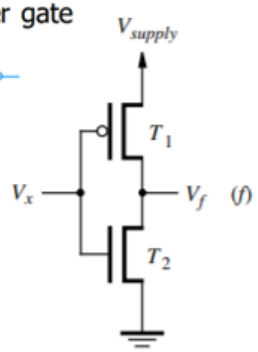
x_1	x_2	T_1	T_2	T_3	T_4	f
0	0	on	on	off	off	1
0	1	on	off	off	on	1
1	0	off	on	on	off	1
1	1	off	off	on	on	0

(b) Truth table and transistor states

TO MAKE AN OR GATE CONNECT A NOR WITH AN INVERTER GATE



Inverter gate



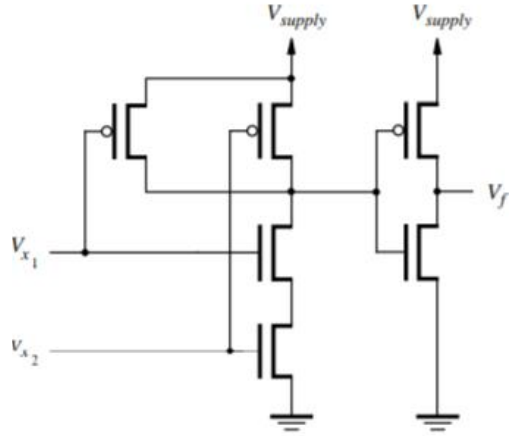
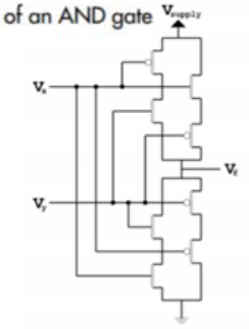


Figure A.19 CMOS realization of an AND gate

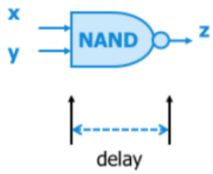
x	V_x	T_1	T_2	V_f	f
0	low	on	off	high	1
1	high	off	on	low	0

(b) Truth table and transistor states



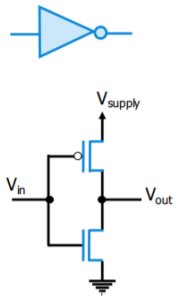
XOR-gate

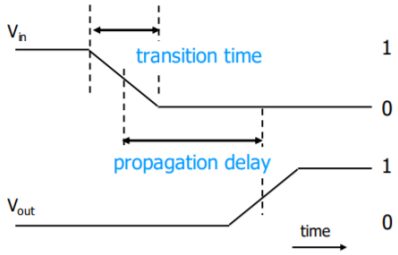
PROPAGATION DELAYS



Switching transistor states takes time (and energy).

Simplest example: inverter





Every network of gates has delays. **The speed of the circuit depends on the maximum number of logic gates that a signal needs to propagate through**. The optimized sum of products is implemented as digital logic.

Still, an **AND gate is more efficient to implement as a NAND & NOT** than strictly making the AND Pullup and Pulldown networks from their literal Minterms. So **not all logic gates delay the same**.

The **number of inputs** to a logic gate is called its **fan-in.**

The **number of branches** coming for next gates it's called **fan-out.**

The **thumb rule** is to **keep fan-in and fan out bellow 10.**

Clock Hertz frequency calculation from:

$$\frac{Propagation\ Delay\ ns * Number\ of\ Gates}{1 - stability\ \%}$$

1 ns = 10^-9 seconds = 10 MHz

COMBINATORIAL CIRCUITS

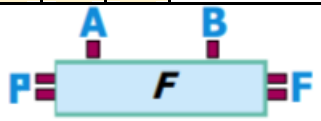
Constructing a **separate circuit for every function** is very **uneconomical**

The goal is to *combine* **Integrated (logic) Circuits** that can compute **different functions** by **taking instruction parameters**:

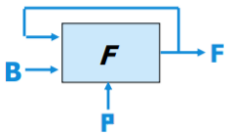
For 4 instructions (2^2), we need 2 instruction bits (p1 = 2nd bit, p0= 1st bit)

2 instruction bits & 2 operand bits			A B				
f(p1,p0,A,B)		p1 p0	00	01	10	11	result type
Add	A XOR B	00	00	01	01	00	signed int
Multiply	A AND B	01	00	00	00	01	signed int
Compare	A - B	10	00	11	01	00	signed int
Or	A + B	11	00	01	01	01	boolean

Combinatorial circuits depend directly and Only on the given input.



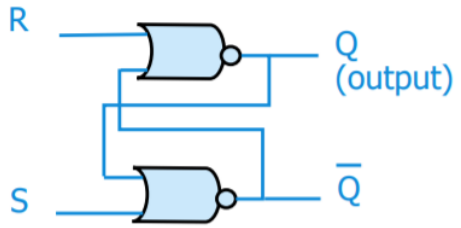
SEQUENTIAL CIRCUITS



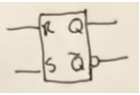
- Depends on input parameters
- Depends on internal state (last result), "recursive"
- Stores prev data. **Counter:** B = 1 and $F_n = F_{n-1} + B$

THE SR LATCH (STORE)

Electronic element that can **store binary information**



Characteristic table			
S	R	Q_{next}	Action
0	0	Q	hold state
0	1	0	reset
1	0	1	set
1	1	X	forbidden

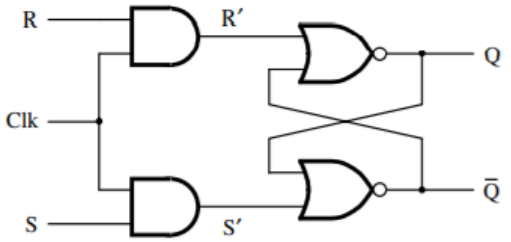


You can set to 1, reset to 0 or keep the current state (neither set nor reset).

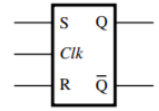
Set and reset at the same time (11) is nonense. You want either, not both.

It would lead to Q=~Q=0 and if you "release" SR back to 00 one path will randomly be faster and feed the other NOR gate and Q will vary from 11 over time. If 11 is never used, we can remember the previous states of R and S based on the current output. **You can also make NAND SR Latches.**

GATED SR LATCH (ENABLED)



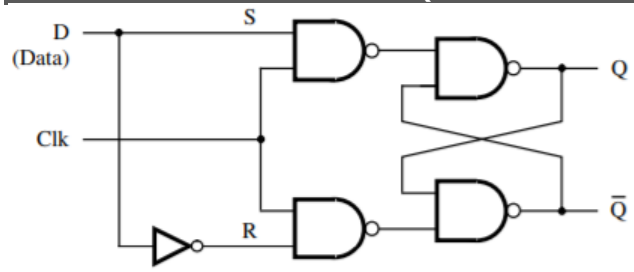
Clk	S	R	Q(t + 1)
0	x	x	Q(t) (no change)
1	0	0	Q(t) (no change)
1	0	1	0
1	1	0	1
1	1	1	x



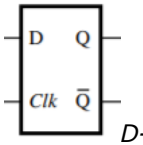
It is the same as the SR latch, but changes in Q_t states have a fixed tempo controlled by a binary clock. Only when clock is 1 Q_t may be updated.

Clocks are essential in logic circuits to properly time the update of variables.

GATED D LATCH (ONE D AND ENABLED)



Clk	D	Q(t + 1)
0	x	Q(t)
1	0	0
1	1	1

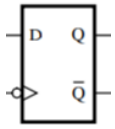


D-latch

A single D input "samples" (sets state) of SR when clock is high and stores/latches (keeps state) when clock is low". *Clk allows multiple changes*

Latch: To retain whatever output state resulted from a previous input signal until reset by another signal.

EDGE TRIGGERING (FLIP-FLOPS)



A flip-flop is edge triggered if the output is *only updated at pulses*:

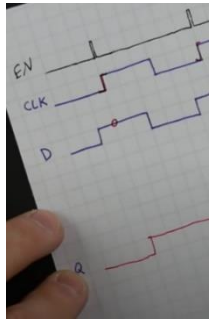
Positive (leading): when clock "jumps" from 0 to 1.

Negative (trailing): when clock "drops" from 1 to 0.

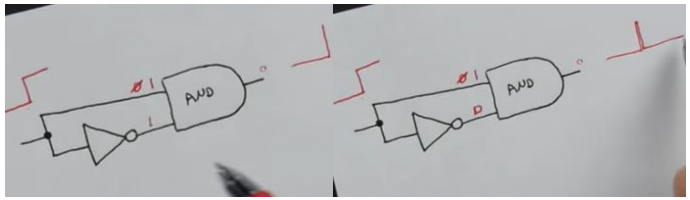
The edge triggered flip-flop is distinguished by the |> under D.

D FLIP-FLOP (POSITIVE EDGE TRIGGERED)

Is a D-Latch that enables changes only at the [+**edge** start of the clock pulse].

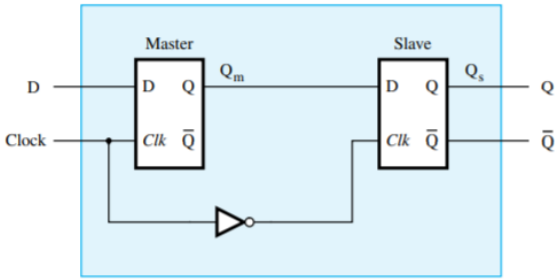


A way to make an +edge detector consists of an AND of 2 opposites where the complement's (opposite) inverter **propagation delay** has for few ns both inputs to be 1.

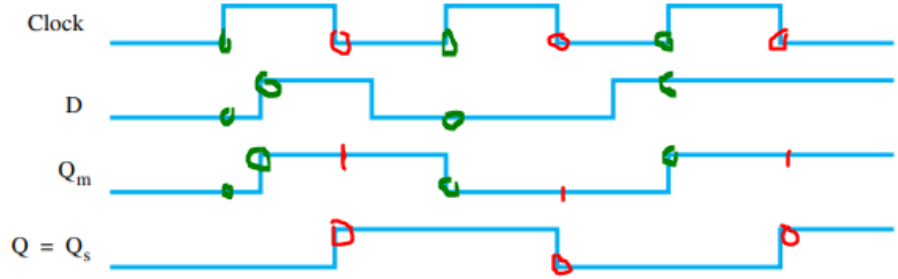


That after the clock will make the circuit +edge triggered.

MASTER-SLAVE D FLIP-FLOP (NEGATIVE EDGE TRIGGERED)



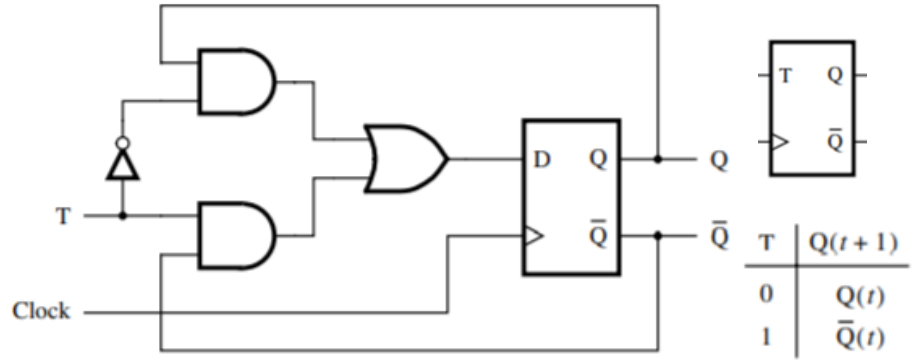
The first D-Latch (master) is enabled (and updated) during clock 1 and transfers the data to the second D-latch that takes the Output of the master as D and waits until the clock drops to 0 to send it. When clock is at 0 the slave doesn't get new Master data, therefore the final output is only produced at clock drops.



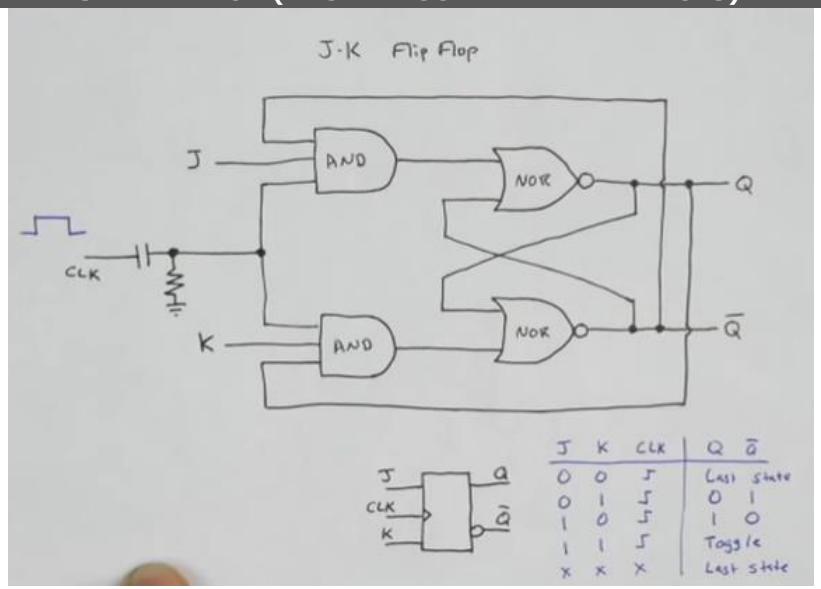
Positive and negative edge triggered D Flip-flops have the same icon.

T FLIP-FLOP

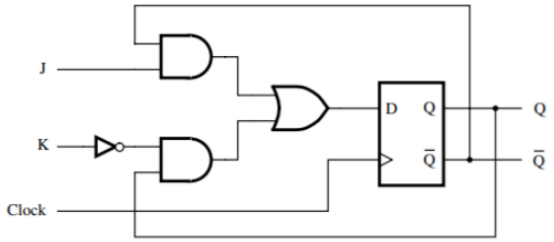
T flip-flop changes the state of its inside D flip-flop every clock cycle if its input T (toggle) is equal to 1.



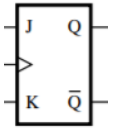
JK FLIP-FLOP (EDGE TRIGGERED AND 2 INPUTS)



Same as SR [(store, but triggered) and it allows JK to be 11 to toggle like T's]

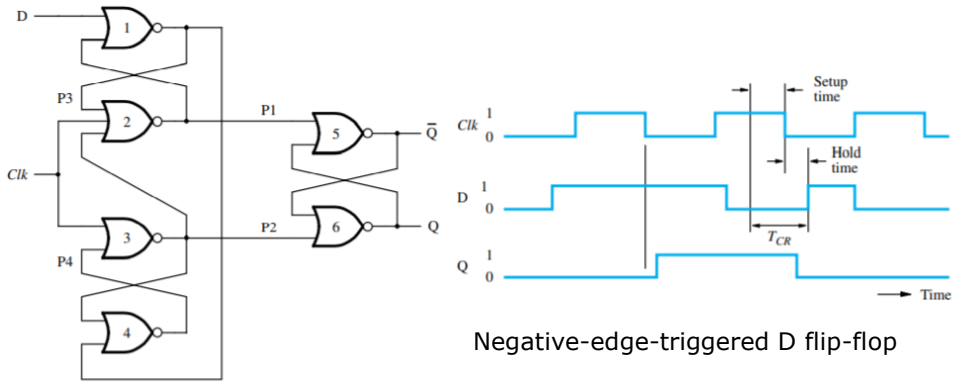


J	K	Q(t + 1)
0	0	Q(t)
0	1	0
1	0	1
1	1	Q̄(t)

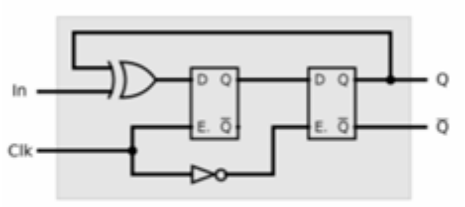


Alternative representation.

EXTRA, OTHER NEGATIVE FLOPS

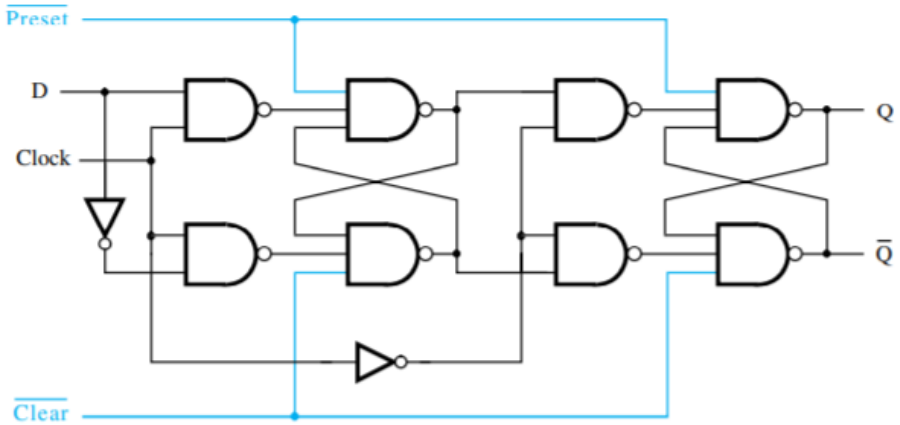


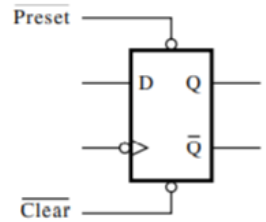
Negative-edge-triggered D flip-flop



Master-slave that feeds its result in XOR (addition) "IN" stands for "inverse". So if IN = 1 and Q = 0, Q becomes 1. If IN = 1 and Q = 1, Q becomes 0. If IN = 0, $Q_t = Q_t$

PRESET AND CLEAR FLIP-FLOPS





Sometimes it is desirable to force a flip-flop into a particular known state (rather than random), especially at PC start**. Preset and Clear are "active low"**, so the opposite holds. If both are 1. The D-latch is controlled by the clock. If clear = 0, it starts at 0, **if preset = 0, it starts at 1**

REGISTERS

Arrangement of a number of D flip-flops synchronized by 1 clock.

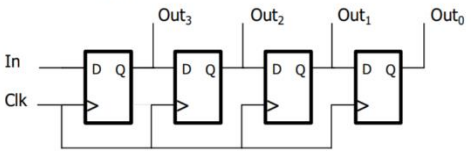
SHIFT REGISTER

Data are written (loaded) into or read from all flip-flops at the same time.

Shift registers moves the output from the utmost left flip-flop to the next one on the right after each pulse and so forth till the far right.

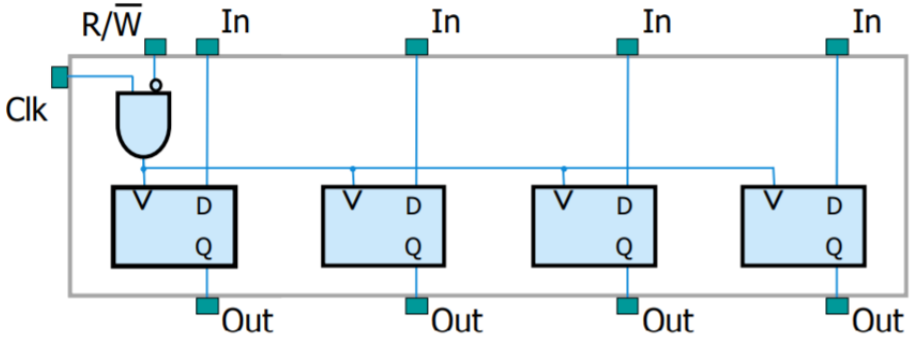
A shift register

- Serial in / parallel out

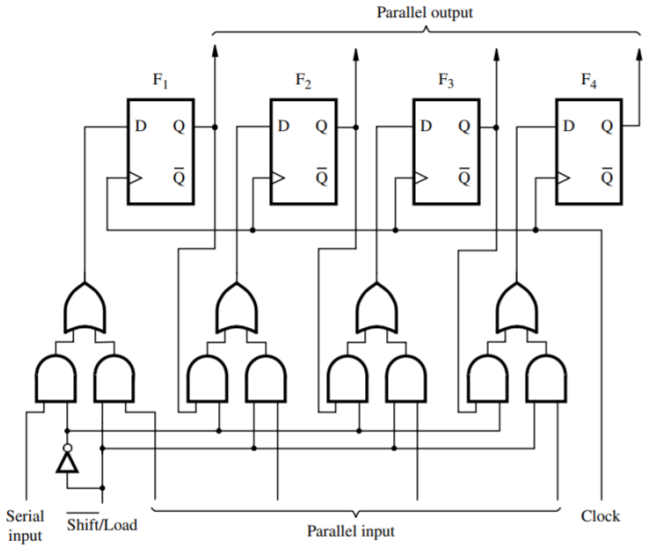


- But delay needs to be accounted for
  - multiple clocks

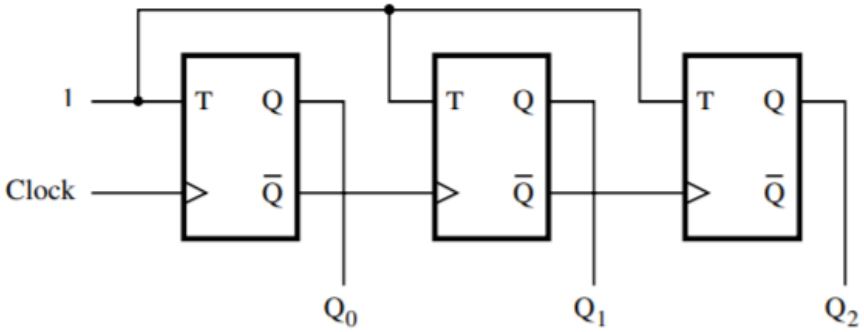
PARALLEL-ACCESS REGISTER



The register clock is controlled by a separate read/write signal (so all states are on hold while on Read, until Write sign). *Alternative form*



COUNTERS (SEQUENTIAL CIRCUIT)



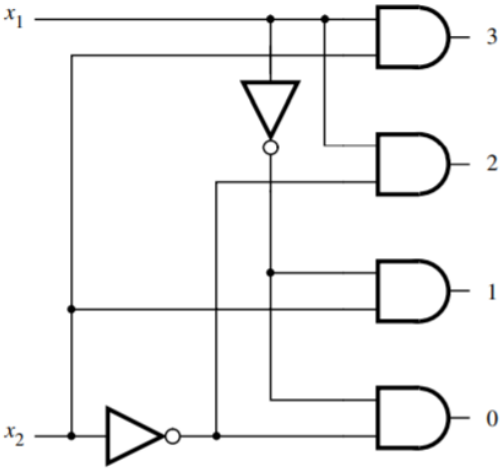
It's like a shift register but instead of shifting the output Q it "enables" the next clock signal when the last flip-flop was 0. (carry over).

A **counter driven by a high-frequency clock** can be used to produce signals whose frequencies are **submultiples of the original clock** frequency. Such a counter is said to be functioning as a **scaler.**

Ripple counter: when the flip-flops are positive edge triggered.

DECODERS (COMBINATORIAL CIRCUIT)

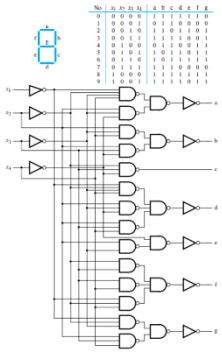
A circuit capable of accepting an n-variable input and generating the corresponding output signal on one out of 2^n output lines.



x_1	x_2	Active output
0	0	0
0	1	1
1	0	2
1	1	3

Figure A.35 A two-input to four-output decoder.

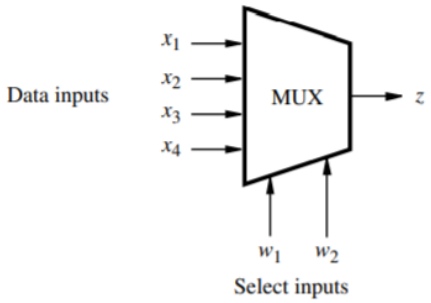
It will allow for easer logic if's circuits rather than if each time you'd have to manually decode it in the next circuit.



Yo could use the binary to decimal decoder to print an electric clock. The one on the right is the BCD to seven-segment display decoder.

MULTIPLEXERS (COMBINATORIAL)

Any one of n data inputs can be selected to appear as the output. The choice is governed by a set of "select" inputs. Such circuits are called multiplexers.



w_1	w_2	z
0	0	x_1
0	1	x_2
1	0	x_3
1	1	x_4

Data inputs could be all of the n = 1MB of memory addresses of a computer, and the multiplexer will easily allow you to select and output a specific byte by just using 20 (2^{20}) select inputs.

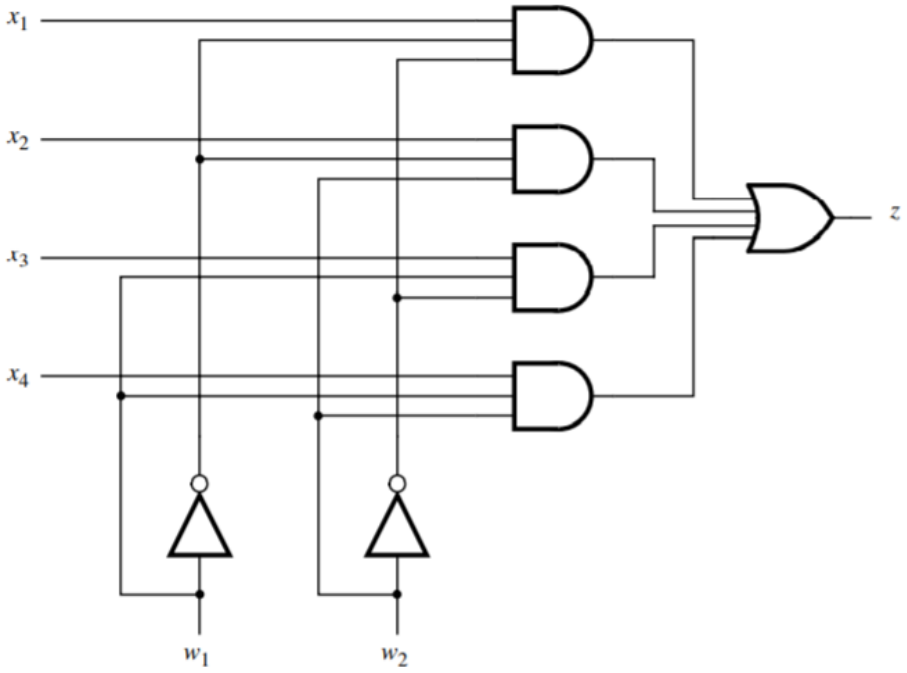
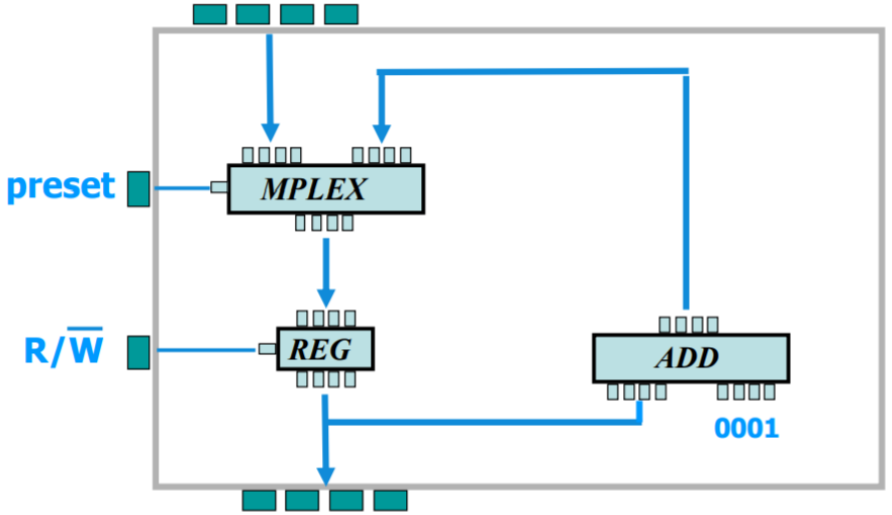


Figure A.37 A four-input multiplexer.

The inputs of a multiplexer will be $\log_2 N$ and the output will be 1.
The inputs of a decoder will be n and the output will be 2^n.
Multiplexers are also used in logic functions.

COUNTER (COMBINATORIAL AND SEQUENTIAL)

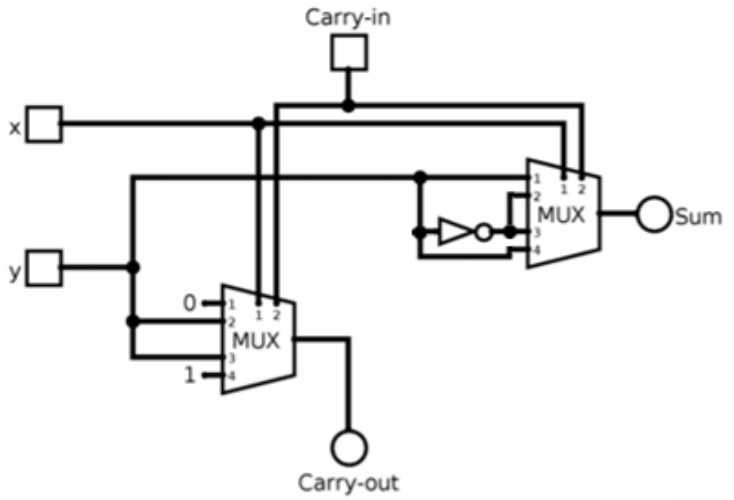


Preset 0/1 tells multiplexer to either output the last register addition or to start the counter at a specific value "V". (input of multiplexer: $\log_2 2 = 1$)

R/W 0/1 read freezes the counter, write resumes it letting the register clock to trigger data transfers from the multiplexor (inputs) to the output pins.

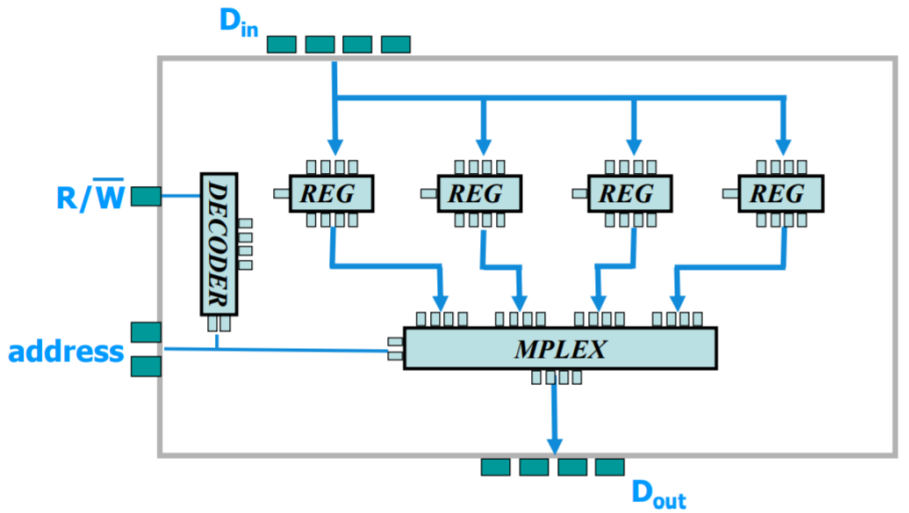
The ADD chips is an XOR circuit.

FULL ADDER WITH MULTIPLEXER



full adder circuit component with only multiplexers

MEMORY

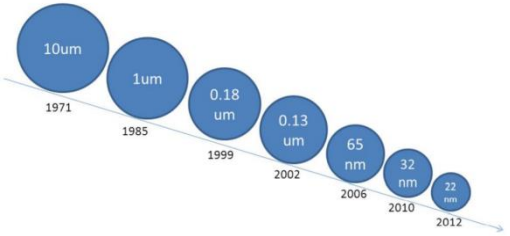


2 address pins to define via the multiplexer one of the 4 registers.

The decoder will take the 2 address bits and send the read/write signal to the chosen register.

MOORE'S LAW

The number of transistors doubles every 1.5-2 years



Current transistor size 5nm

ROCK'S LAW

The cost of semiconductor fabs doubles every 4 years.

Although the trend makes computers cheaper the barriers of entry to produce competitive computers are higher and therefore less parties are involved in making chips (Intel one of them).

WEEK 3 – DATA REPRESENTATION

In the past floating points conversions and the millennium bug has costed lots of money. That's why engineers now set up things for the long term.

RADIX (BASE) TO DECIMAL

$$d_3 d_2 d_1 d_{0_b} = d_3 * b^3 + d_{2*} * b^2 + d_1 * b^1 + d_0 * b^0$$

d_n = n^{th} digit, b = base.

$$3210_{16} = 3 * 16^3 + 2 * 16^2 + 1 * 16 + 0 * 16^0 = 12816$$

DECIMAL TO BASE

Repeatedly subtract the largest power of BASE that fits in the number or repeatedly divide by BASE, the n remainders (n_0 = LSB) form the bit string.

LSB = Least Significant Bit = 0^{th} bit. MSB = Most Significant Bit (often sign).

RADIX A TO > RADIX B (A = B^M)

Large radix to small radix where big radix is a b^m multiple of the smaller:

$$d_A = \blacksquare_{m-1_B} \blacksquare_{m-2} \cdots \blacksquare_{0_B}$$

$$65_8 = ([\__2\ \__1\ \__0]\ [\__2\ \__1\ \__0])_2 \quad (2^3 = 8)$$

$$\underline{110}\ \underline{101}$$

$$6\quad 5$$

Big to small: Split digits of the large radix into m digits of small base.

RADIX B TO < RADIX A (A = B^M)

Small radix to big radix where big radix is a b^m multiple of the smaller:

$$d_A = \blacksquare_{m-1_B} \blacksquare_{m-2} \cdots \blacksquare_{0_B}$$

$$(1010\ 1111\ 1000)_2 = ([\__0]\ [\__0]\ [\__0])_{16} \ (16 = 2^4)$$

$$\underline{1010}\ \underline{1111}\ \underline{1000}$$

$$10(A)\ 15(F)\quad 8$$

Small to big: Group digits of the small radix into m digits of large base.

ASCII Table:

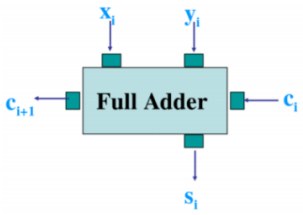
Binary	Oct	Dec	Hex	Glyph
010 0000	040	32	20	sp
010 0001	041	33	21	!
010 0010	042	34	22	"
010 0011	043	35	23	#
010 0100	044	36	24	$
010 0101	045	37	25	%
010 0110	046	38	26	&
010 0111	047	39	27	'
010 1000	050	40	28	(
010 1001	051	41	29	)
010 1010	052	42	2A	*
010 1011	053	43	2B	+
010 1100	054	44	2C	,
010 1101	055	45	2D	-
010 1110	056	46	2E	.
010 1111	057	47	2F	/
011 0000	060	48	30	0
011 0001	061	49	31	1
011 0010	062	50	32	2
011 0011	063	51	33	3
011 0100	064	52	34	4
011 0101	065	53	35	5
011 0110	066	54	36	6
011 0111	067	55	37	7
011 1000	070	56	38	8
011 1001	071	57	39	9
011 1010	072	58	3A	:
011 1011	073	59	3B	;
011 1100	074	60	3C	<
011 1101	075	61	3D	=
011 1110	076	62	3E	>
011 1111	077	63	3F	?

Binary	Oct	Dec	Hex	Glyph
100 0000	100	64	40	@
100 0001	101	65	41	A
100 0010	102	66	42	B
100 0011	103	67	43	C
100 0100	104	68	44	D
100 0101	105	69	45	E
100 0110	106	70	46	F
100 0111	107	71	47	G
100 1000	110	72	48	H
100 1001	111	73	49	I
100 1010	112	74	4A	J
100 1011	113	75	4B	K
100 1100	114	76	4C	L
100 1101	115	77	4D	M
100 1110	116	78	4E	N
100 1111	117	79	4F	O
101 0000	120	80	50	P
101 0001	121	81	51	Q
101 0010	122	82	52	R
101 0011	123	83	53	S
101 0100	124	84	54	T
101 0101	125	85	55	U
101 0110	126	86	56	V
101 0111	127	87	57	W
101 1000	130	88	58	X
101 1001	131	89	59	Y
101 1010	132	90	5A	Z
101 1011	133	91	5B	[
101 1100	134	92	5C	\
101 1101	135	93	5D	]
101 1110	136	94	5E	^
101 1111	137	95	5F	_

Binary	Oct	Dec	Hex	Glyph
110 0000	140	96	60	`
110 0001	141	97	61	a
110 0010	142	98	62	b
110 0011	143	99	63	c
110 0100	144	100	64	d
110 0101	145	101	65	e
110 0110	146	102	66	f
110 0111	147	103	67	g
110 1000	150	104	68	h
110 1001	151	105	69	i
110 1010	152	106	6A	j
110 1011	153	107	6B	k
110 1100	154	108	6C	l
110 1101	155	109	6D	m
110 1110	156	110	6E	n
110 1111	157	111	6F	o
111 0000	160	112	70	p
111 0001	161	113	71	q
111 0010	162	114	72	r
111 0011	163	115	73	s
111 0100	164	116	74	t
111 0101	165	117	75	u
111 0110	166	118	76	v
111 0111	167	119	77	w
111 1000	170	120	78	x
111 1001	171	121	79	y
111 1010	172	122	7A	z
111 1011	173	123	7B	{
111 1100	174	124	7C	|
111 1101	175	125	7D	}
111 1110	176	126	7E	~

Dec	Hex	Oct	Bin
0	0	000	0000
1	1	001	0001
2	2	002	0010
3	3	003	0011
4	4	004	0100
5	5	005	0101
6	6	006	0110
7	7	007	0111
8	8	010	1000
9	9	011	1001
10	A	012	1010
11	B	013	1011
12	C	014	1100
13	D	015	1101
14	E	016	1110
15	F	017	1111

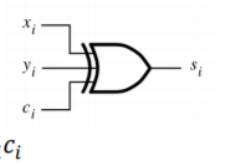
Binary Addition		
1		carry
	1	x
	1	y
1	0	answer



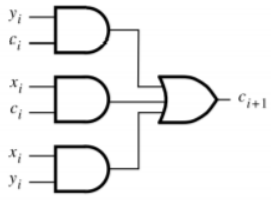
FULL ADDER

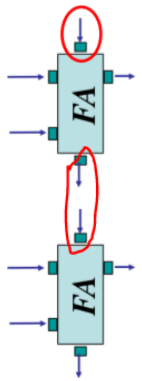
The sum and carry bits

- Sum:
  - $s_i = xi \oplus y_i \oplus c_i$
  - $s_i = x_i\overline{y}_i\overline{c}_i + \overline{x}_i y_i\overline{c}_i + \overline{x}_i\overline{y}_i c_i + x_i y_i c_i$



- Carry:
  - $c_{i+1} = y_i c_i + x_i c_i + x_i y_i$





Concatenate full adders to add larger numbers. **Carries ripple through**

BINARY CODED DECIMAL (BCD)

1	8	3	7
0001	1000	0011	0111

It's a decimal number whose digits are represented in 4 bit binary. **Pros**: each 4bit group can be decoded to a digital led **number display**. Cons: **5 bits are not used.**

SIGN & MAGNITUD, ONE'S COMPLEMENT, TWO'S COMPLEMENT

$b_3b_2b_1b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+7	+7	+7
0 1 1 0	+6	+6	+6
0 1 0 1	+5	+5	+5
0 1 0 0	+4	+4	+4
0 0 1 1	+3	+3	+3
0 0 1 0	+2	+2	+2
0 0 0 1	+1	+1	+1
0 0 0 0	+0	+0	+0
1 0 0 0	−0	−7	−8
1 0 0 1	−1	−6	−7
1 0 1 0	−2	−5	−6
1 0 1 1	−3	−4	−5
1 1 0 0	−4	−3	−4
1 1 0 1	−5	−2	−3
1 1 1 0	−6	−1	−2
1 1 1 1	−7	−0	−1

For all systems: if MSB = 1 it's negative if MSB = 0 it's positive.

All positive are the same.

S&M negative: MSB multiplies unsigned number by -1.

S&M 0: It will also multiply 0 by -1 (-0), so 2 zeros.

S&M range: $[-(2^{n-1}-1), 2^{n-1}-1]$

1C negative: flip zeros and 1s

1C 0: 2 zeros (-0 = 111…1)

1C range: same as S&M

2C negative: 1C negative + 1; **zero :** only +0; **range:** : $[-(2^{n-1}), 2^{n-1}-1]$

2C addition = subtraction. Just add numbers and ignore the last carry.

Negative overflow if + and + yield -, Positive overflow if – and – yield + (number exceeds the number of bits and it gets "truncated").

Addition of two different sign numbers will never yield integer overflow.

EXCESS-X

Offset of -x. 0000 = -x. 0001 = 1-x. **Range:** $[-x, 2^{n-1}-x]$. Offset in decimal unless specified. Used as exponent in floats.

(Multiplication and addition in binary is the same as in decimal)

SIGN EXTENSION

- Extension: Represent **n**-bit as **k**-bit number
- If n < k:
  - (Unsigned) 100 -> **0..0**100
  - (S&M) s10 -> s**0..0**10
  - (1C, 2C) x10 -> **x....x**10
- If n > k:
  - (U, S&M) If **n-k** MSB bits (excluding sign) zero, **remove** leftmost **n-k** bits
  - (1C, 2C) If **n-k+1** MSB bits equal, **remove** leftmost **n-k** bits
  - Else, overflow thus impossible

Can conversion between **decimal** and **binary** numbers always be done exactly for

1. integers? Yes
2. fractions? **No**
   - Counter-example:
   - $0.2_{10} = 0.125_{10} + 0.0625_{10} + ...$
     $0.2_{10} = 0.\mathbf{0011}0011001100....._2$

x	2^{-x}
0	1
1	0.5
2	0.25

FIXED POINT FRACTIONS

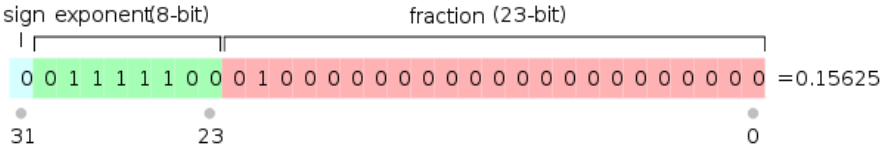
bit	2^{bit}
-1	0.5
-2	0.25
-3	0.125
-4	0.0625
-5	0.03125
-6	0.015625
-7	0.0078125
-8	0.00390625
-9	0.001953125
-10	0.000976563

FLOATING POINT NUMBERS (IEEE-754)

Real number binary representation that assigns different meanings to certain bits so that the user can pass the sign, the exponent and the mantissa of a number. Changing the exponent allows the decimal point to "float". Float = sign * base^{exponent} * mantissa (like scientific notation).

Mantissa = units digit coefficient followed by the dot and the decimals

IEEE-754 float: Mantissa is always 1.something, therefore the 1. is skipped



bit sign, exponent excess-127, fraction is fixed point binary MSB start at -1

Floating-point numbers that are inverted differ only in sign bit (S&Mish)

Exponent all 0s is makes mantissa implicit 1. become 0. (making 0 possible)

Exponent all 1s is for infinite if fractions bits are all 0 otherwise NaN i.e. 0/0

Decimal to float: 1 create fixed point binary 2 Shift binary places until 1.xx 3 Number of shifts + 127 = exponent 4 remove 1. from mantissa. Fill in.

WEEK 4 – INSTRUCTION SET ARCHITECTURE (ISA)

An ISA is an **abstract** model of a computer that serves as the **interface** between software and hardware. Which specifies: supported data types, what state there is (main memory and registers) and their semantics (consistency and addressing modes) instruction set, I/O model.

Multiple ISA implementations due to performance, size, and cost constraints

COMPARISION CRITERIA

Flexibility: Complexity of what the ISA can do (op and address modes)

Programmability: Complexity for programmers (code length, #registers)

Implementability: …for hardware (encoding of instr, memory consistency)

CISC (COMPLEX INSTRUCTION SET COMPUTER)

- **Stack** based;
- **Register** based;
- **Long instruction word** (assembly code line);
- explicit **parallelism**;
- Minimal instruction Set Computer (**MISC**);
- **Traditional** architectures (legacy);
- **Powerful** instructions (complex and too many);
- **Memory to memory** operations. 68xxx and x86 family

PROS:

- **Easier to program**;
- **Reduced code** size;
- Complexity in hardware;
- Legacy (politics), its in our **PCs and servers**.

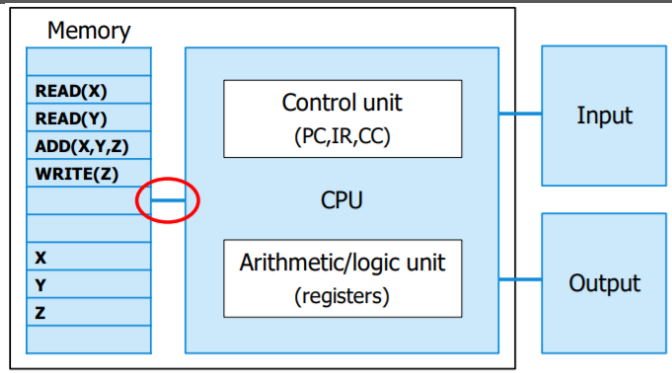
CONS:

- Instruction **encoding complexity**
  - **variable-length** instructions,
  - **many addressing modes**,
- **Slower** than RISC (stack is slower than registers),
- Consumes **more energy** (not in embedded systems, portable devices),
- many **unused and too specific instructions**).

RISC (REDUCED INSTRUCTION SET COMPUTER)

- **Reduced Instruction Set**
- Small number of instructions
- **Load/Store from Memory**
- Operations between **registers**
- **Large** register file
- Present in: PowerPC, ARM

VON NEUMANN ARCHITECTURE



MEMORY ORGANIZATION

Bits (D flip-flops) **grouped into words** (parallel access registers) **grouped into Memory chips** (or SoC (system on a chip)).

Addressable by **byte**: 1 bit too little useful, a word too cumbersome.

1 byte (8 bits) = an ASCII character. Sweet spot.

In **x64 architecture:**

word = 2 bytes, **long** (doubleword) = **4 bytes**, **quad = 8 bytes**

Alignment

- Accessing items from memory
  - size matters (amortize setup)
  - alignment matters too (less circuitry)
- Alignment
  - address is multiple of item size
  - bin size must be a power of 2
- Unaligned access **may** be supported

Q: any disadvantages?

Variable length

1600	POP R1	POP R2
1602	MUL R1, R2	
1604	ADD VAR, R2	
1606	PUSH R2	CALL
1608	foo	POP R1
1610	...	
1612		
1614		

⇧ word unaligned

Stack Memory Addresses are 8 bytes because x64 operates on quadwords (64 bits). Data must fit into those quads. Aligned access is easier for the programs to push/pop things around (like an Ikea store). Often empty bits

BIG ENDIAN VS LITTLE ENDIAN

Big = Left to right (IBM, The Internet), **Little !=** "Unit group" Right to left but the **contents of the group are still read Left to Right.** (Intel)

TYPES OF INSTRUCTIONS (INTEL VANILLA)

Data Copy Operations

- Between **memory and registers**: [R1] <- M(LOC)
- Between **memory locations**: $M(LOC_1)$ <- $M(LOC_2)$
- Between **registers**: $[R_1]$ <- $[R_2]$

Arithmetic and **logic** operations:

- Add / - / * / % / divide… [R1] <- [R1] + [R2]

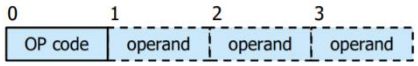
Flow control operations:

Branch_IF_[R1]>[R2] LOOP

I/O operations:

ISA LAYOUT IN MEMORY

- An instruction may span multiple words
- #bits/specifier may be different per instruction type



- Format: INSTR operand*
- Meaning: An instruction with 0 or more operands where an operand could be a register, constant, memory address

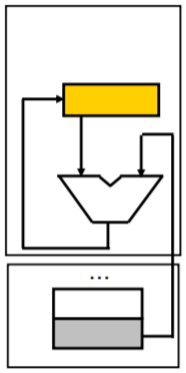
INSTRUCTION ADDRES FORMAT

#address	Operand 1	Operand 2	Operand 3	Pitfall
0 (stack)	Pop destination	(But slides insist its 0)		
1(accumulator)	Destination (implicit acc)			
2 (M-M) (R-M)	Source	Destination		Overwrite
3 (M-M-M) (R-R-R)	Source 1	Source 2	Destination	Long in

EVOLUTION OF INSTRUCTION SETS

Name	Date	Addresses	Example
Accumulator	<1960	1	68HC11
Stack	1960-1970	0	
Memory-Memory	1970-1980	2/3	
Register-Memory	1970-today	2	CISC
Register-Register	1960-today	3	RISC

ACCUMULATOR (SINGLE REGISTER)



Simple design: easy to implement and program

Memory is bottleneck: can't keep frequently accessed data in the processor

- 16-bit words
  - byte addressable
- Data
  - S&M integers
- Instruction
  - 4-bit opcode
  - 12-bit address (m)

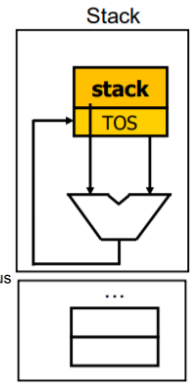
Load A
Add B
Store C

An implicit operand (register) called the **Accumulator**

```
[Accum] ← M(A)
[Accum] ← [Accum] + M(B)
M(C) ← [Accum]
```

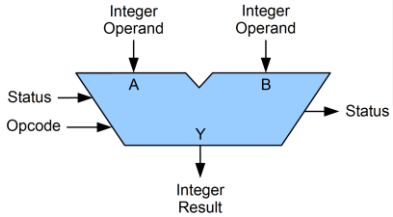
STACK

- Example:
  - Push A `[TOS] ← M(A)`
  - Push B `[TOS] ← M(B)`
  - Add `[TOS] ← [TOS] + [TOS-1]`
  - Pop C `M(C) ← [TOS]`
- Note: implicit references to stack for ALU ops

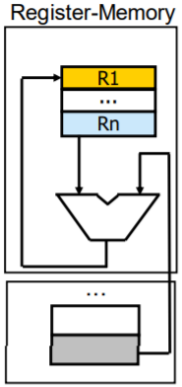


ALU: Aritmetic and Logic Unit (MPLXER)

TOS: Top of the stack



REGISTER-MEMORY (CISC)



- **16 General purpose registers**
  - aka register file
  - growing over time
- **faster than memory**
- **fewer address bits**, so **easier to encode**
- **Memory is bottleneck**
  - can't keep frequently accessed data in the processor

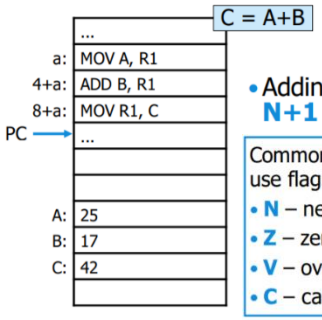
R0 reserved for 0

Add A,B
`M(B) ← M(A) + M(B)`

Addressing modes:

- Direct
- Register
- Immediate
- Index

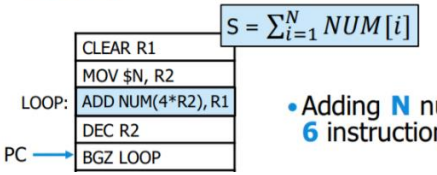
Straight-line sequencing



- Adding **N** numbers takes **N+1** instructions

Common branch instructions use flags in the status register
- **N** – negative
- **Z** – zero
- **V** – overflow
- **C** – carry

Looping



- Adding **N** numbers takes **6** instructions

X86-64 INSTRUCTION FORMAT

CISC-style: at most (!) 15 bytes

- Legacy prefixes (1-4 bytes, optional)
- Opcode with prefixes (1-4 bytes)
- ModR/M (1 byte, if required)
- SIB (1 byte, if required)
- Displacement (1/2/4/8 bytes, if required)
- Immediate (1/2/4/8 bytes, if required)

<op>
0x0F <op>
0x0F 0x38 <op>
0x0F 0x3A <op>

ADDRESSING MODES (INTEL VANILLA)

Immediate	value
Direct	M(value/location)
Register	[reg]
Register Indirect	M([reg])
Base with displacement	M([reg] + disp)
Index with displacement	M([reg]*s + disp)
Base with index	M([reg1] + [reg2]*s)
Base with index and displacement	M([reg1] + [reg2]*s + disp)

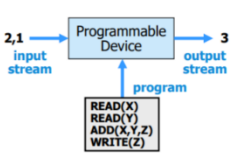
Scale s=1,2,4 or 8 disp= 8, 16, 32 or 64-bit *signed* number

Base with index and displacement: **Base = starting memory address**, **scale = bytes in word** (increment between addresses), **index "array pointer"**, displacement = "column" byte displacement within the word?

WEEK 1 – ASSEMBLY

X86-64 (IA-64), AT&T SYNTAX WITH GCC IN LINUX

- Logic circuits
  - parameterized circuits
- Programmable devices
- Central Processing Unit
  - IA-32
  - IA-64
  - PowerPC
  - ARM
  - ...



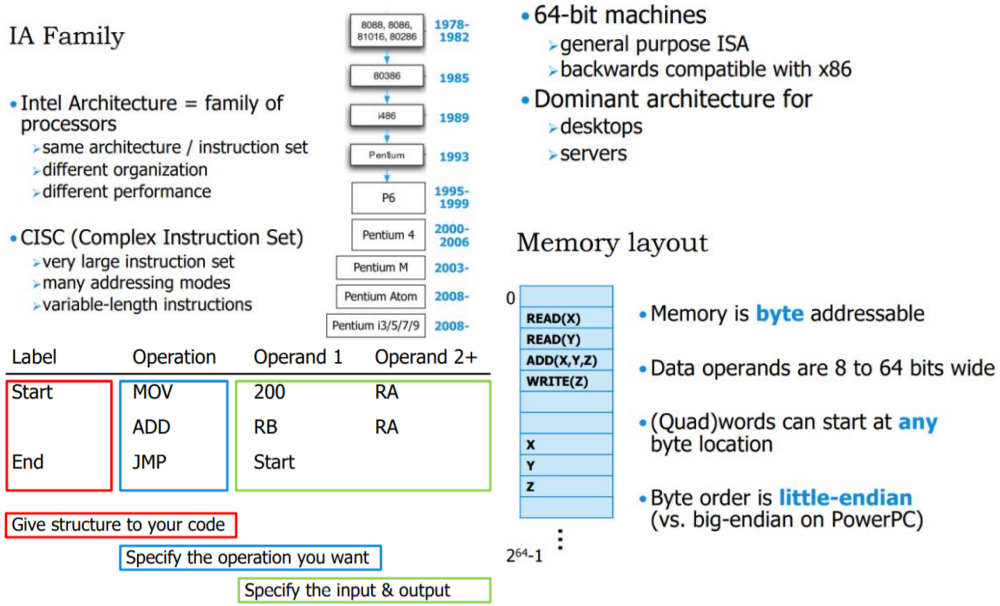
Program creation and execution flow

- Type source program
  - `vim foo.s` → editor → ASCII file
- Translate to machine code
  - `gcc foo.s` → assembler → object file
- Load image & execute
  - `./a.out` → load & run → image

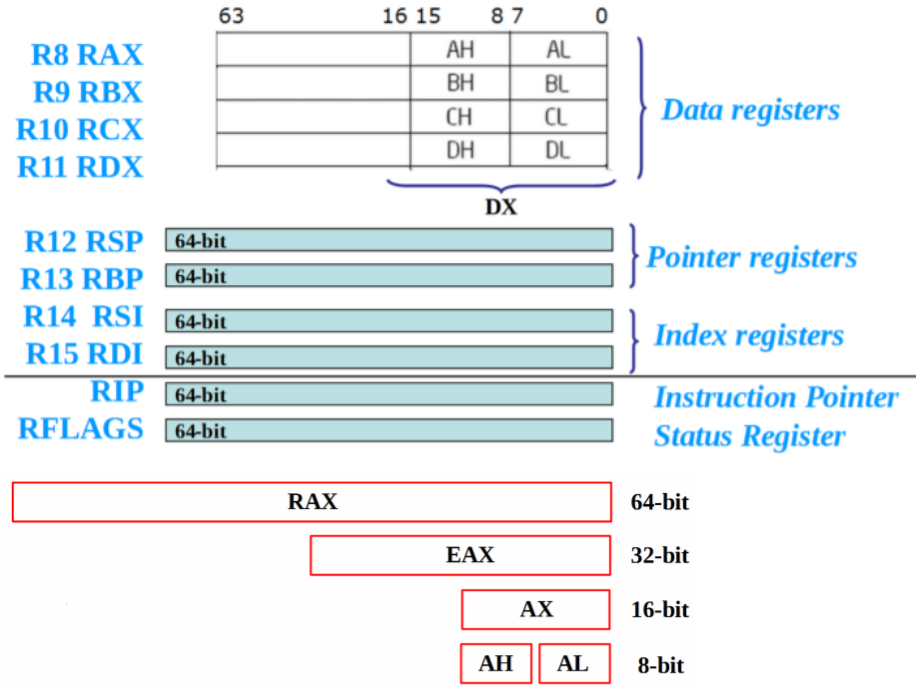
A **symbolic notation** for machine language

- Improves Readability:
  - Machine code: 0010 1101 1001 0001
  - Assembly: Move RA, SUM
- Access to all hardware resources
  - Required for writing e.g. a bootloader
- Achieves greater performance
  - for critical applications

ASSEMBLY STRUCTURE OF INTEL ARCHITECTURE



GENERAL PURPOSE REGISTERS



64-bit	Lower 32 bits	Lower 16 bits	Lower 8 bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

Other important registers:

RIP = instruction pointer, points to the next instruction to be executed. changing this register is the same as a jumps

RFLAGS = register that stores information about the last calculation (flags) to use for conditional jumps

Variable-length instructions 1-15 bytes

Format for move and arithmetic: INSRT SRC, DST

Format for comparison: CMP OPRND1, OPRND 2

Format for Flow Control: JMP LOCATION

opcode	operands	function	description
mov	src,dst	dst = src	copy
push	dst	(%rsp) = dst, %rsp -= 8	pushes a value onto the stack
pop	src	%rsp += 8,src=(%rsp)	pops a value off the stack
xchg	A,B	A,B = B,A	switches the contents of A and B
addq	src,dst	dst = dst + src	adds src to dst
subq	src,dst	dst = dst – src	subtracts src from dst
inc	dst	dst = dst + 1	adds 1 to dst
dec	dst	dst = dst – 1	subtracts 1 from dst
mulq	src	rdx:rax = rax * src	multiplies rax by src (UNSIGNED)
imulq	src	rdx:rax = rax * src	multiplies rax by src (SIGNED)
divq	src	rdx:rax = rax / src	divides rax by src (SIGNED)
idivq	src	rdx:rax = rax / src	divides rax by src (SIGNED)
jmp	label		jumps to label (unconditional)
je	label		jumps to label (if equal)
jne	label		jumps to label (if not equal)
jg	label		jumps to label (if greater than)
jl	label		jumps to label (if less than)
jle	label		jumps to label (if less than or equal)
jge	label		jumps to label (if greater than or equal)
call	label	push <current 15ddress + 1>, jmp label	calls a function
ret		jmp (%rsp)	returns to caller
loop	label	dec %rcx, jnz label	
cmp	A,B	A – B (answer not stored but flags set)	compares 2 numbers. Jump instruction follows
xorq	src,dst	src = src xor dst	bitwise xor
orq	src,dst	src = src and dst	bitwise and
andq	src,dst	src = src or dst	bitwise and
shlq	A,dst	src = src << A	shift left
shrq	A,dst	src = src >> A	shift right
not	dst	dst = 1111111- dst	bitwise inversion of dst
neg	dst	dst = 0 – dst	2’s complement, result of not and add 1
leaq	A, dst	dst = &A	load effective 15ddress (& means 15ddress of)
int	int_no		software interrupt (see linux system calls above, used together with int 0x80)

ADDRESSING MODES (AT&T)

example	name	description
movq $25,%rax	immediate	Loads the decimal value into rax
movq $label,%rax	immediate (pointer)	loads the location of the label into rax
movq label,%rax	direct	loads the quadword at the location of the label into rax
movq (%rbx),%rax	indirect	loads the quadword at the location pointed to by rbx into rax
movq 8(%rbx),%rax	indirect offset (positive)	loads the quadword 8 after the location pointed to by rbx into rax
movq -8(%rbx),%rax	indirect offset (negative)	loads the quadword 8 before the location pointed to by rbx into rax
movq (%rbx,%rcx),%rax	indirect variable offset	loads the quadword at %rcx after the location pointed to by rbx into rax
movq (%rbx,%rcx,8),%rax	indirect variable scaled offset (negative)	loads the quadword at %rcx*8 after the location pointed to by rbx into rax
movq 8(%rbx,%rcx,8),%rax	indirect variable scaled offset (negative) +constant	loads the quadword at 8 after %rcx*8 after the location pointed to by rbx into rax

MOVB	Move one byte	8 bit
MOVW	Move one word	16 bit
MOVL	Move one double word	32 bit
MOVQ	**Move one quad word**	**64 bit**

Scale s=1,2,4 or 8 disp= 8, 16, 32 or 64-bit signed number

movzb move 0 extended byte.

ASSEMBLER DIRECTIVES

directive	explaination
.quad	reserves space for a 64 bit number to be stored
.long	reserves space for a 32 bit number to be stored
.word	reserves space for a 16 bit number to be stored
.byte	reserves space for a 8 bit number to be stored
.asciz	reserves space for a string of text to be stored, automatically terminated by a 0 (NULL)
.ascii	reserves space for a string of text to be stored, *not* automatically terminated by a 0 (NULL)
.skip n	skips n bytes. useful for defining arrays of data. This should normally only be used in the .bss
.equ	defines symbolic names for expressions (i.e. constants)

Nibble = 4 bits (not an assembler directive)

ASSEMBLER SECTIONS

The .**text segment** is intended to hold all instructions. The .text segment is read-only. It is perfectly fine to include **constants and ASCII strings** in this segment.

The **.data segment** is used for initialized variables (variables that receive an initial value at the time you write your program, **such as those created with the .word directive**).

The .**bss segment** is intended to hold **uninitialised variables** (variables that receive a value only at runtime). Therefore, this section is not part of the executable file after compilation, unlike the other two sections.

(Not a section): **.global label** makes label visible to other programs. The **main label** must be exported because the operating system needs to know where to start running your program.

X86 CALLING CONVENTION

The calling convention (System V AMD64 ABI) that is used on *nix systems is as follows. for 64 bit programs only The first six integer or pointer arguments passed in the registers in this order:

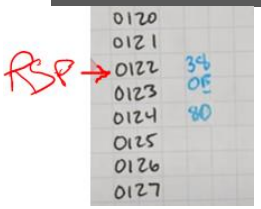
1. RDI
2. RSI
3. RDX
4. RCX
5. R8
6. R9

(with sometimes R10 as a static chain pointer in case of nested functions)

Additional arguments are to be passed on to the stack

The **return values are stored in RAX** (In case of a 64 bit number) and in RDX:RAX (MSB:LSB) in case of 128 bit numbers.

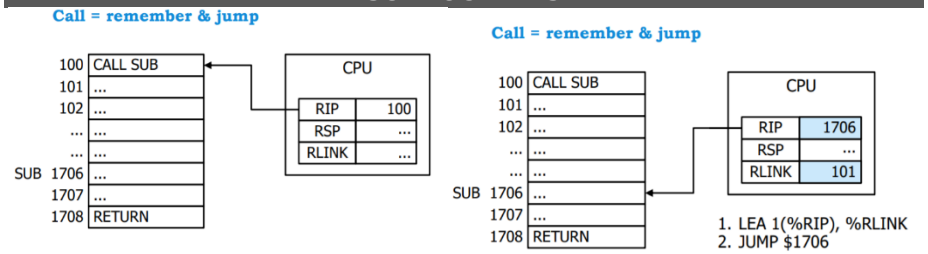
THE STACK



You can push to the top or pop (take out) from the top of the stack. **Pushing values** makes the RSP (register stack pointer) **jump a smaller address** and populate it with such value. If you want to collect the last second push value you type 8(%rsp) 8 being the scale of the byte addresses. In 64 architecture the jumps are 8.

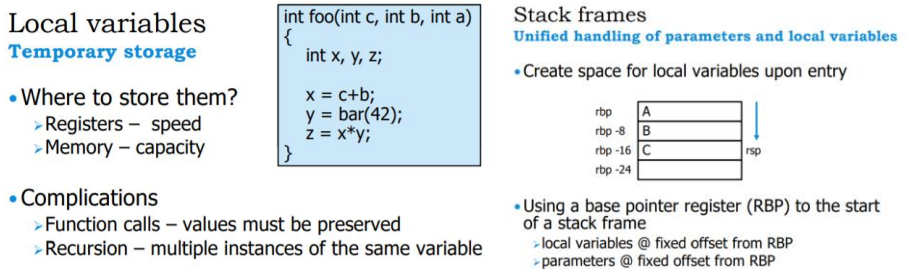
Popping values would remove the value from the top of the stack and assign it to a source and the RSP will return to its last place.

SUBROUTINES

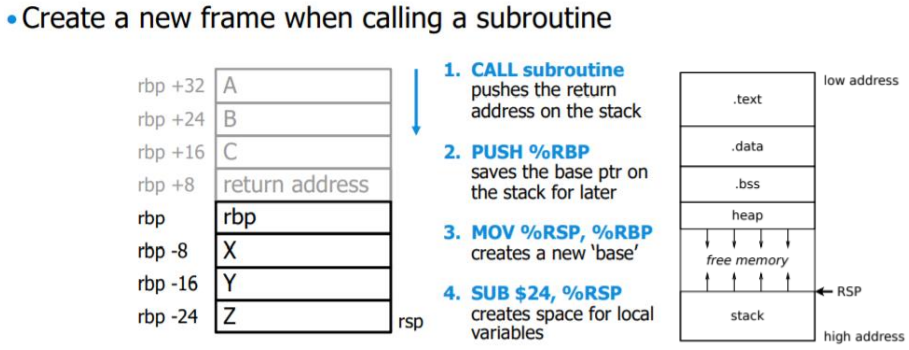


Calling a subroutine will make a jump and remember the next instruction address for after the return.

You can pass parameters through registers (hold on to calling conventions) or you can push them on the stack (hard to keep RSP offsets).



Prologues and epilogues with RSBP can help you keep separate stack frames for each subroutine.



GDB DEBUGGER

GDB is a debugger which can help find segfaults or find other mistakes in your program. to use it compile it using the -g option (put it directly after "gcc") and then instead of running it like ./<programname>, you run it as gdb ./<programname>. this should launch you into a gdb environment. in this environment you can use the following commands:

b n (or breakpoint). this sets a breakpoint on line n

print code. this prints whatever you specify in code. this can be a full c expression, or a register name (e.g. $rdi or $rax)

x/nx p print n 32 bit words after p. p can be an adress or register. this is useful for reading whats on the stack (e.g. x/10x $rbp)

n (or next) steps ahead one instruction. when it finds a function call it will not step into instructions inside this function. useful to skip large functions like c stdlib function like printf

s (or step) steps ahead one instruction. this one does go into large functions

r (or run) runs the program until the next breakpoint or the end

c (or continue) after a breakpoint, continue restarts execution like run did until it encounters another breakpoint or the program ends. useful if a breakpoint is in a loop and you want to go to the next iteration

start starts the program, places a breakpoint on line one so you can imediately start using s and n

when using GDB your program must be compiled with -g and your code must be in a .text section

CHAPTER 2 AND 9 EXTRA NOTES

Unsigned integers uses less bits than Excess because excess still represents negative side when 000000

Exponent of the IEEE-754 has to be within -126,127 - 128 yields inf or Nan

000xxxxxxx: **3**-bit & **7**-bit instructions.

Max would be 2^**3** – 1 + 2^(**7**-**3**). You can only have 1 opcode at a time…

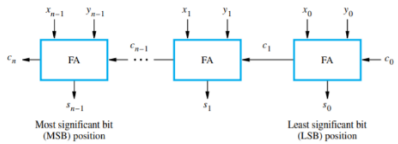
CISC vs RISC assembly difference is RISC uses 3 operands in math and it only accepts registers.

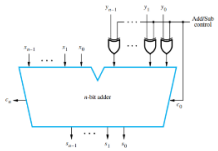
CISC doesn't need to LOAD things into registers.

Vanilla uses LOAD for moving into a REGISTER and STORE R, (SP) for PUSH. Store seems to be the only one that has reversed order. (store src, dst)

Loading to a memory is MOVE.

x_i	y_i	Carry-in c_i	Sum s_i	Carry-out c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1





EXAM PREP

BYTE CONVERSION

Byte = 8 bits (2^3)
KiB = 2^10 bytes
MiB = 2^20 bytes
GiB = 2^30 bytes
TiB = 2^40 bytes
PiB = 2^50 bytes
EiB = 2^60 bytes
ZiB = 2^70 bytes
YiB = 2^80 bytes

FLOATING POINT ADDITION

1. Identify the operand with the smaller exponent
2. Make the smaller exponent equal to the larger
3. Compensate by adding zeros to the binary fraction (don't forget implicit 1 of the mantissa)
4. Add both mantissas
5. Present the sum in a float number with the high exponent.

FLOAT CREATION

Prof. dr. Hook is tasked with improving the new floating point standard created by the CSE1400 team last year to be used for storing grades efficiently. The improvements should make sure that a grade between 1 and 10 (inclusive) can be represented exactly to a quarter grade point, for example: 9.75, 8.25, 5.75.

How many bits should he pick for the sign bit, mantissa, and exponent? Assume the exponent is represented as an unsigned integer.

A. 1 sign bit, 4 mantissa bits, 2 exponent bits
B. no sign bit, 5 mantissa bits, 2 exponent bits
C. no sign bit, 4 mantissa bits, 2 exponent bits
D. 1 sign bit, 4 mantissa bits, 3 exponent bits

Solution: To represent numbers with a quarter of a unit precision exactly, we need to be able to represent 2^{-2}. Additionally, we will need to be able to go up to 9.75 with the same precision. 9.75 can be represented exactly as $(1001.11)_2$. To write this number as an IEEE-754 number, we need to shift the decimal point 3 places to get $(1.00111)_2 \cdot 2^3$. This part after the decimal point shows that a mantissa of at least 5 bits is required. The exponent has to be at most 3, so 2 bits will be enough. No sign bit is required as grades are always a positive number.

Positive overflow: $\overline{a_3} \cdot \overline{b_3} \cdot s_3$; **Negative overflow:** $a_3 \cdot b_3 \cdot \overline{s_3}$

Positive integers overflow, negative integers overflow

Which of the following statements is **true** about an IEEE-754 32-bit number?

A. $1/0$ is not representable as an IEEE-754 number.
B. $2^{64} \cdot 2^{-64}$ is not representable as an IEEE-754 number.
C. $2^{64} \cdot 2^{64}$ is not representable as an IEEE-754 number.
D. $2^{100} + 2^{100}$ is not representable as an IEEE-754 number.

This causes an overflow as the represented exponent range is $(-126, 127)$ and the total exponent will be 128.

MULTIPLEXER ADDER (TRUTH TABLE TRICK)

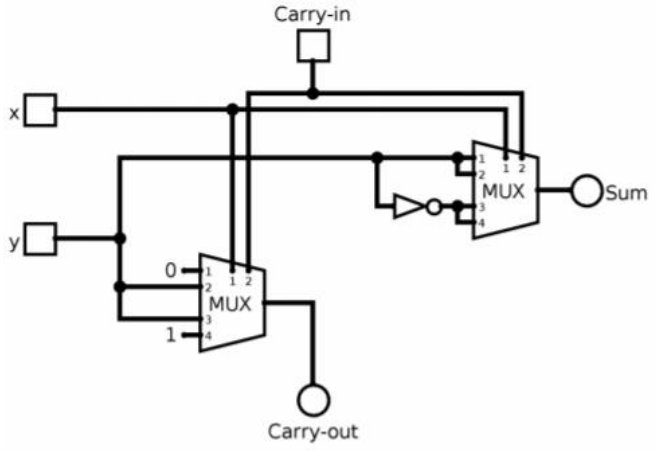


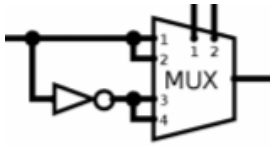
Figure 3: Boole's circuit, containing a mistake

André-Marie Ampère and George Boole are participating in the annual Circuit Olympics (CO). One of the challenges is to build a full adder circuit component with only multiplexers and NOT gates. Boole comes up with a design in the blink of a CPU cycle and proudly shows it to Ampère. However, Ampère replies with laughter, as he immediately spots a mistake in Boole's design. What should change in Boole's design (shown in Figure 3) to be a correct full adder?

A. Switch the select inputs for the multiplexer that outputs the sum
B. Switch the select inputs for the multiplexer that outputs the carry-out
C. Switch data inputs 1 and 3 for the multiplexer that outputs the sum
D. Switch data inputs 2 and 4 for the multiplexer that outputs the sum

Solution: The solution can be verified by (1) knowing the truth table for both output signals and (2) checking if the circuit implementation is valid using the technique of Figure A.38 from the book.

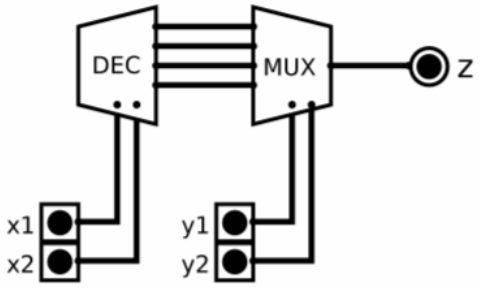
How to solve: Assume all x, y, (and carry) data inputs are 0 and ignore the select inputs. What would these incoming arrows input into the multiplexer? If the incoming y was 0, then you'll get the 1234 truth table of:



0
0
1
1

But the actual sum truth table is 0110.
Therefore we know 2nd and 4th must be swapped.

MULTIPLEXOR Z OTPUT FORMULA



w_1	w_2	z
0	0	$\sim x_1 * \sim x_2$
0	1	$\sim x_1 * x_2$
1	0	$x_1 * \sim x_2$
1	1	$x_1 * x_2$

$z = \sim y_1 * \sim y_2 * \sim x_1 * \sim x_2 + \sim y_1 * y_2 * \sim x_1 * x_2 + y_1 * \sim y_2 * x_1 * \sim x_2 + y_1 * y_2 * x_1 * x_2$

BOOLEAN SIMPLIFICATION

$$(c)\ x_1\bar{x}_2 + \bar{x}_2x_3 + x_3\bar{x}_1 = x_1\bar{x}_2 + x_3\bar{x}_1$$

x1~x2 + ~x2x3 + ~x1x3 = x1~x2 + x1~x2x3 + ~x1~x2x3 + ~x1x3 = x1~x2(x3 + 1) + ~x1x3(~x2 + 1) = x1~x2 + ~x1x3

Explanation:

~x2x3 can be separated into x1~x2x3 + ~x1~x2x3, since ~x2x3 = ~x2x3(~x1 + x1) = x1~x2x3 + ~x1~x2x3.

Then we combine the terms again.

1 + anything = 1 in boolean algebra, so the terms x3 and ~x2 can be omitted there.

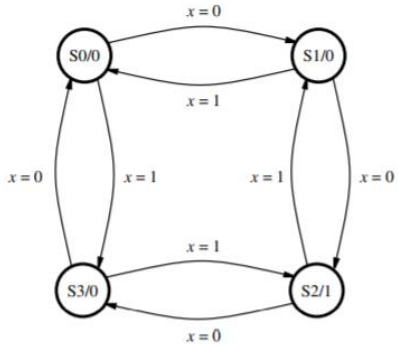
KOOMEY'S LAW

Koomey's law describes a trend in the history of computing hardware: for about a half-century, the number of computations per joule of energy dissipated doubled about every 1.57 years

BOOLEAN LAWS

Name	Algebraic identity	
Commutative	$w + y = y + w$	$wy = yw$
Associative	$(w + y) + z = w + (y + z)$	$(wy)z = w(yz)$
Distributive	$w + yz = (w + y)(w + z)$	$w(y + z) = wy + wz$
Idempotent	$w + w = w$	$ww = w$
Involution	$\overline{\overline{w}} = w$	
Complement	$w + \overline{w} = 1$	$w\overline{w} = 0$
de Morgan	$\overline{w + y} = \overline{w}\,\overline{y}$	$\overline{wy} = \overline{w} + \overline{y}$
	$1 + w = 1$	$0 \cdot w = 0$
	$0 + w = w$	$1 \cdot w = w$

MOD-4 UP/DOWN COUNTER



Present state	Next state $x=0$	Next state $x=1$	Output z
y_2y_1	$Y_2\ Y_1$	$Y_2\ Y_1$	
0 0	0 1	1 1	0
0 1	1 0	0 0	0
1 0	1 1	0 1	1
1 1	0 0	1 0	0

Figure A.46 State diagram of a mod-4 up/down counter that detects the count of 2.

Present state	Next state $x=0$	Next state $x=1$	Output z
S0	S1	S3	0
S1	S2	S0	0
S2	S3	S1	1
S3	S0	S2	0

Figure A.47 State table for the example of the up/down counter.

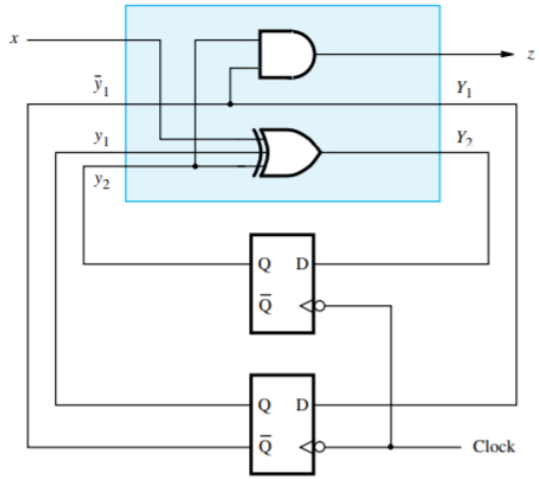


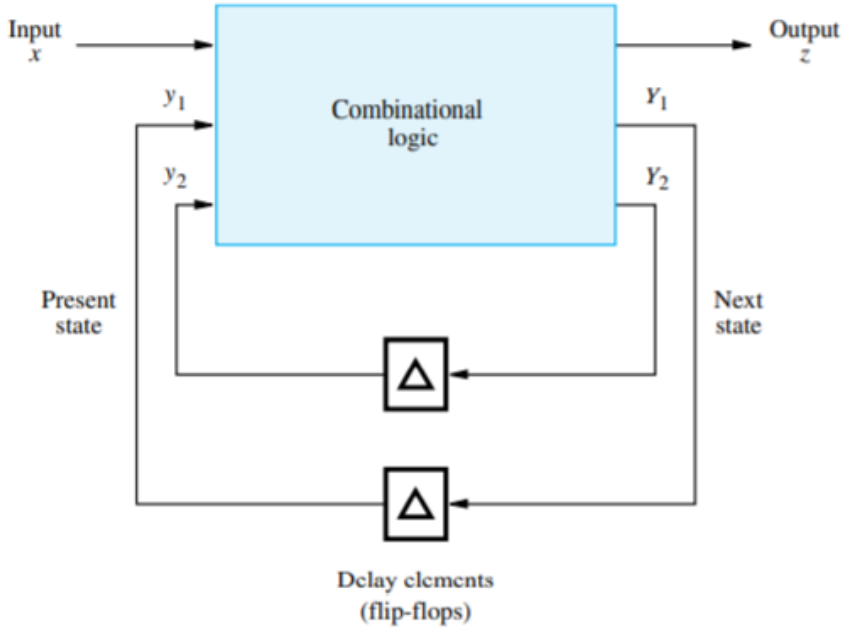
Figure A.49 Implementation of the up/down counter.

The output z is determined as

$$z = y_2\bar{y}_1$$

These expressions lead to the circuit shown in Figure A.49.

FINITE STATE MACHINE



1. Develop an appropriate state diagram or state table.
2. Determine the number of flip-flops needed, and choose a suitable type
3. Determine the values to be stored in these flip-flops for each state in the state diagram. This is referred to as state assignment.
4. Develop the state-assigned state table.
5. Derive the next-state logic expressions needed to control the inputs of the flip-flops. Also, derive the expressions for the outputs of the circuit.
6. Use the derived expressions to implement the circuit.

TYPES OF COMPUTERS

Embed computers = for specific purpose in embed systems, industrial

Personal computers = consumer market and variety of purposes

Servers and Enterprise systems = network of large computers with DB

Super computers and grid computers = high performance, expensive, demanding computations (i.e. weather forecasting), grid = high speed network of combination of personal computers, cloud is emerging trend

MEMORY TYPES

PRIMARY

Also called main memory. Electronic. Programs are stored here. It has distinct addresses (byte addressable usually) It includes RAM, which can be accessed at a fast fixed time.

CACHE MEMORY

Faster than RAM, holds sections of the program currently being executed. At start of program is empty, data fetched from main memory is copied here to interact with the CPU (cache is usually inside cpu close to registers).

SECONDARY STORAGE

Magnetic, optical and flash memory devices that keep data even when there's no power.

ARITHMETIC AND LOGIC UNIT

Any arithmetic or logic operation, such as addition, subtraction, multiplication, division, or comparison of numbers, is initiated by bringing the required operands into the processor, where the operation is performed.

PARALLELISM

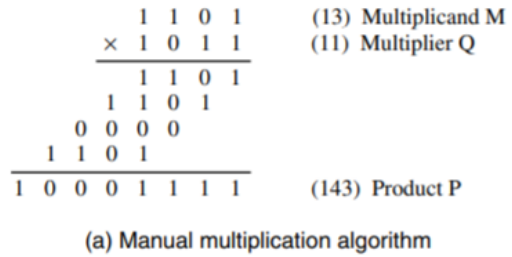
You can gain better performance (i.e. run code faster)

By doing parallel tasks, using processors with multiple cores, or using multiple processors (and or a combination of everything).

PROBLEMS CHAPTER 1

Page 46

MULTIPLICATION



(a) Manual multiplication algorithm

You just copy the operand 1 or return n 0s if the other operand's bit is 0 and then offset the n number each time one bit further.

You end up with n-1 additions with n being the number of digits of one of the operands.

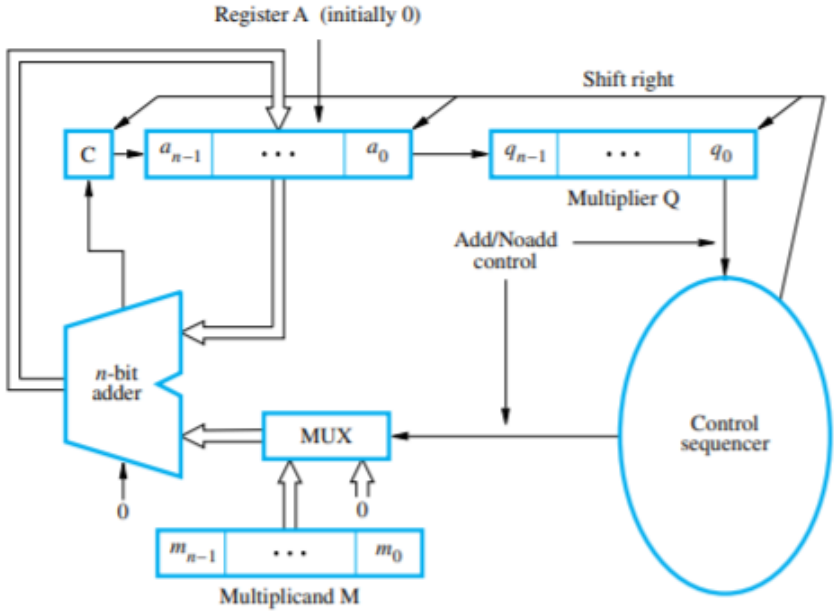
Unless the operands have the same bit length, there are two possible number of sums.

The **Sequential Circuit Multiplier** seems to use n number of additions.:

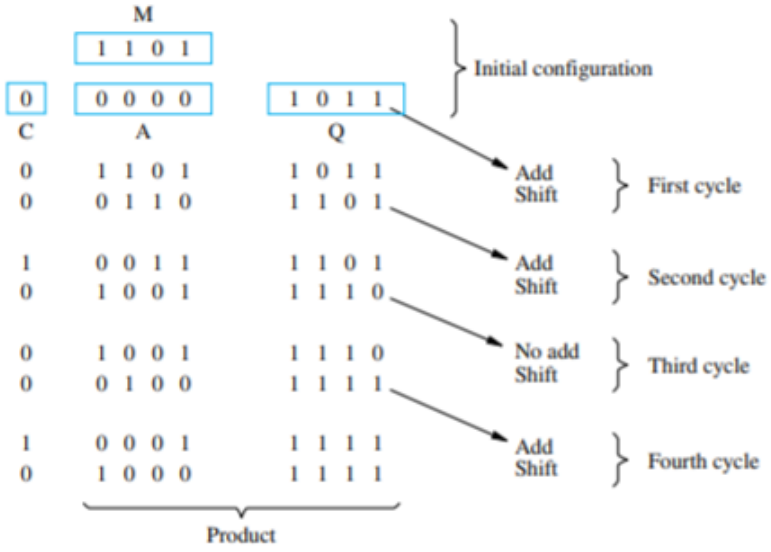
This circuit performs multiplication by using a single n-bit adder n times.

BOOTH ALGORITHM

It handles both positive and **negative** multipliers uniformly. Second, it achieves some efficiency in the number of additions required when the multiplier has a few large **blocks of 1s**.



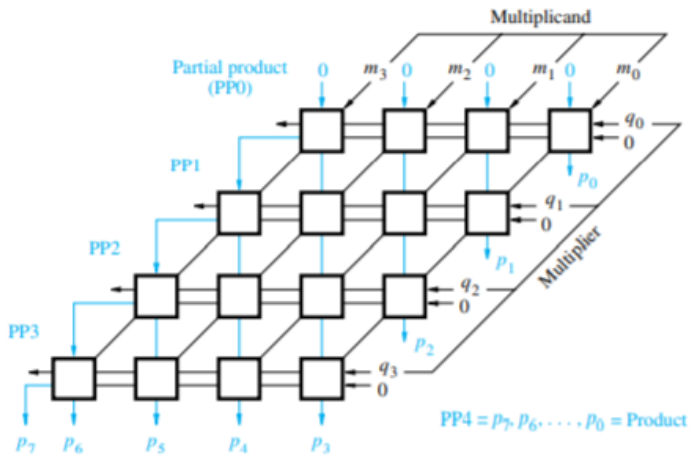
(a) Register configuration

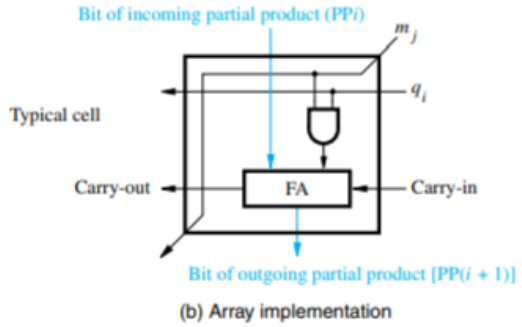


(b) Multiplication example

Figure 9.7 Sequential circuit binary multiplier.

I promise that I will not use unauthorized help from people or non-course materials during my exam. I will create the answers on my own and I will create them only during the allocated exam time slot. I will not provide help to or ask help from other students during their exam.

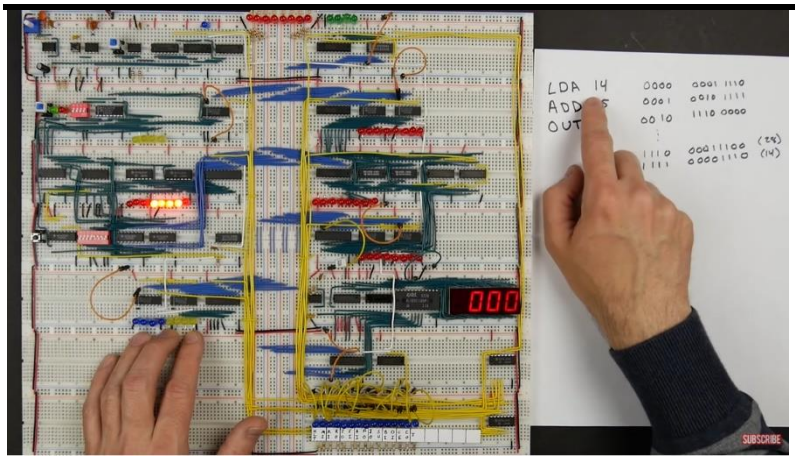




(b) Array implementation

Figure 9.6 Array multiplication of unsigned binary operands.

WEEK 6 – CPU LOGIC (BEN EATER INTRO)



Assume we have a made up Assembly language with the instructions:

LDA SRC (A) – Loads the parameter implicitly into register A

ADD SRC (A) – Adds the parameter and saves it into register A

OUT (A) – Outputs the contents of register A into the decimal display

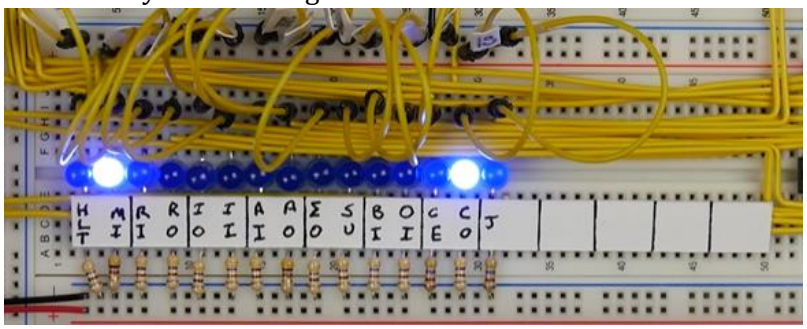
However, these instructions that we just made up don't tell the computer anything at all yet. We decided that the instructions of our assembly language are of fixed length.

4 bits for the operation code and 4 bits for the paramater (whether the parameter is interpreted as a memory location or as an immediate value is up to the "micro routine" to decide).

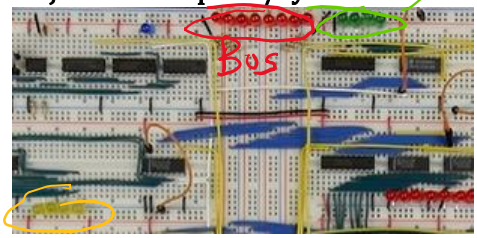
For now, let's assume that LDA has opcode 0000, ADD has opcode 0001 and OUT has opcode 1110.
Let's also assume that we started to store the program at memory location 0000, the second line of code of that program is at 0001 and the last one at 0010.

How do we execute the program above?

1. The **program counter** keeps counting/jumping to the next line of code (command) that needs to be fetch and sent to the **instruction register**. At the start of the program, the program counter will be the 1st command.
2. The first thing that is going to happen when we start to execute a program (or command of a program) is that we need to **load the contents of the (first) command from the memory (via the memory data register)** and put it in the **Instruction Register** (it tells us which command (opcode) we are currently running). This is the start of the **fetch cycle**: we fetch the instruction from memory and put it into the instruction register.
3. In order to get the contents of memory location that the program counter indicates, first we need to take the value of **the program counter** and move it to the **Memory Addres Register** (to indicate which memory address cpmtemts to fetch).
4. Every opcode instruction will then start with a fetch cycle, that goes throu the following control logic micro instruction (via to the Bus):
   a. $ProgramCounter_{out}$, $MemoryAddresRegister_{in}$: Which means the counter outputs to the bus and the Memory Address Register reads from the bus.

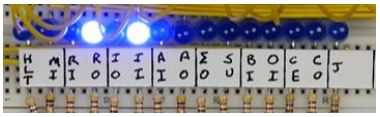


So for the progam above first instruction LDA 14, we did $ProgramCounter_{out}$, $MemoryAddresRegister_{in}$ **in just 1 clock pulse/cycle**

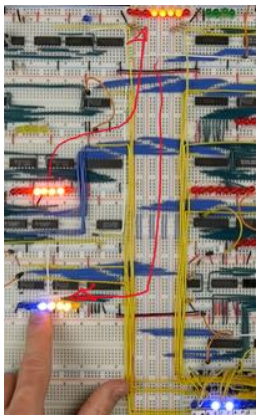


$MI = MemoryAddresRegister_{in}$
$CO = ProgramCounter_{out}$

   b. In the **next clock pulse/cycle** wee need to move the contents of Memory Address Register into the Instruction register: RAM_{out}, $InstructionRegister_{in}$



$RO = RAM_{out}$
$II = InstructionRegister_{in}$



Nothing happens until the clock pulse/cycle is completed. Which lasts long enough to have enough time to setup the control logic without conflicts and provide the desire output.
Ben Eater's instruction register is purposefully 2 colored, the Most Significant 4 bits expect the opcode and the 4 bits from the right expect the operand.

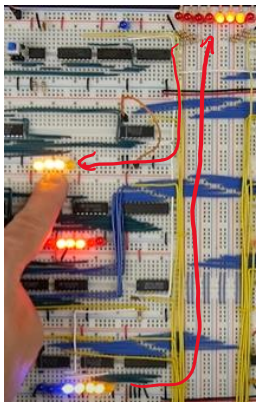
c. **The last part of the fetch cycle is to increment the program counter**, so that it will be pointing to the next instruction we would like to load: $ProgramCounter_{Enable}$ (CE in Ben's Computer)

Therefore, we need to make sure that the control logic hardware circuits know how to do:

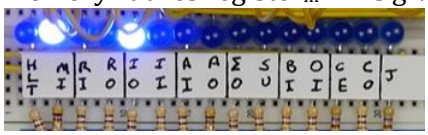
1. $ProgramCounter_{out}$, $MemoryAddresRegister_{in}$
2. RAM_{out}, $InstructionRegister_{in}$
3. $ProgramCounter_{Enable}$

Now that we've finally fetched the instruction. We will execute it.

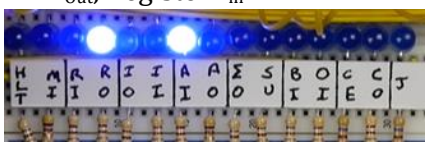
From here, first we read the 4 most significant bits of the instruction register. Which is the opcode: Since the 4 MSB = 0001 =LDA instruction, it is just moving the operand to register A. Such a thing requires us to first update the Memory Address Register with the 4 LSB (operand that points memory address) so that we can Read the values from that memory address and eventually load them in A:



d. Purposefully, only the least significant bits of the Instruction Register are connected to the bus. Therefore $InstructionRegister_{out}$ will output the operand to the bus. Consequently, we want to update the Memory Addres Register to get the bus contents, so $MemoryAddresRegister_{in}$. This gives: $InstructionRegister_{out}$, $MemoryAddresRegister_{in}$



e. Now we want to take the contents of the Ram, and move them to Regiser A:
RAM_{Out}, $RegisterA_{in}$

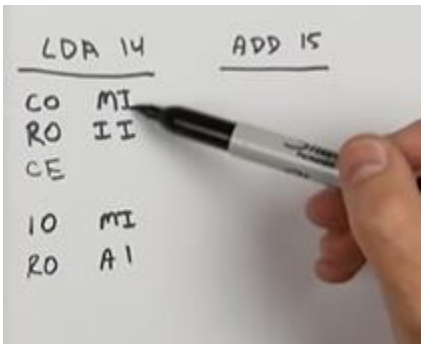


Each of these micro instructions (d and e) required 1 clock pulse/cycle.

Therefore, the single LDA 14 operation has been exectued with the following micro instructions (CPU Logic):

1. $ProgramCounter_{out}$, $MemoryAddresRegister_{in}$
2. RAM_{out}, $InstructionRegister_{in}$
3. $ProgramCounter_{Enable}$
4. $InstructionRegister_{out}$, $MemoryAddresRegister_{in}$
5. RAM_{Out}, $RegisterA_{in}$

A total of 5 clock cycles.



The next instruction (and all instructions) start the same way:

Program Counter spits out the Next Memory Address location, and the Memory Location Address Registers listens to the bus and updates its contents.
The RAM spits out the contents of the instruction, and the Instruction registers takes in those contents.

Before executing the opcode, the program counter gets incremented
Then it depends on the opcode (and your hardware). What happens next. Ben's computer ALU uses Register A and Register B as operands. So in this case you'll have to do IO MI, RO B1. In ben's computer ALU's output is in E, so next is EO AI

8-bit CPU control logic (Ben Eater) https://www.youtube.com/watch?v=dXdoim96v5A

BPU - CHAPTER 5 (BASIC PROCESSING UNIT = BPU = CPU); RISC STYLE

The processing unit executes machine-language instructions and coordinates the activities of other units in a computer. Such as fetching, decoding and executing such instructions. The processing unit is often called the central processing unit CPU. The term central is not as appropiate today as it was in the past because today's computers often include several processing units. **processor** is a synonym for **processing unit** and CPU.

Processors that operate in parrallel have a **pipeline**d organization where the execution of an instruction is started before the execution of the precedeing instruction is completed.
Superscalar operation is to fetch and start the execution of several instructions **at the same time**.

FUNDAMENTAL CONCEPTS

The processor fetches one instruction at a time and performs the operation specified. These instructions are fetched from successive memory locations unti la branch or a jump instruction is encountered.

The processor uses the program counter (PC) to keep track of the addres of the next instruction to be fetched and executed. After fetching the instruction, the program counter is updated to point to the next instruction in sequence. A branch instrucion may cause the PC to not (automatically) increase by 1 but by the address of the jump.
When an instruction is fetched it is placed in the instruction register, from where it is interpreted or decoded by the processor's control circuitry.

RISC-style steps for executing instructions:

1. Fetch the contents of the memory location pointed by the PC and load them into the IR (**instruction fetch phase**).

   $$IR \leftarrow [[PC]]$$

2. Increment the PC to point to the next instruction.

   $PC \leftarrow [PC] + k$ where k is the integer that denotes the byte difference between address1 and address2

3. Carry out the operation specified by the instruction in the IR (**instruction execution phase**). Which generally consists of one or more of the following actions:
   a. **Read** the contents of a memory location **and load them into a processor register**
   b. **Read** data from one or more processor registers
   c. Perform an arithmetic or logic operation and place the result ino a processor register
   d. **Store** data from a processor register into a memory location

You LOAD registers and STORE in memory.

The processor communicates with memory through the processor-memory interface.
The instruction address generator updates the PC after each instruction is fetched

The register file is a memory unit that contains the general purpose registers
THe ALU does the computations, whose computations are stored in a rigster in the register file (For RISC) (Z in CISC)

INSTRUCTION EXECUTION

Load R5, X(R7) //DST, SRC

Which uses Index Addressing mode to load a word of data from memory location X + [R7] into register 5. By doing:

1. Fetch instruction from the memory
2. Increment program counter
3. Decode instruction to determine the operation to be performed
4. Read register R7
5. **Add the Immediate value X to the contents of R7** (extra step)
6. Use the sum as the effective address of the source operand. and read the contents of that location
7. Load the data received from that location into register R5

5 STEP RISC INSTRUCTIONS

Depending on the hardware some operations can be done at the same time. Book assumes 5 steps for RISC processor.

Load R5, X(R7)

1. Fetch the instruction and increment the program counter
2. Decode the instruction and read the contents of register R7 in the register file
3. Compute the effective addres X + [R7]
4. Read the **memory** source operand
5. Load the operand into the destination register R5

Add R3, R4, R5, //DST, SRC, SRC

1. Fetch the instruction and incerement the program counter
2. Decode the instruction and read the contents of source register R4 and R5
3. Compute the sum [R4] + [R5]
4. Load the result into the destination register R3

However, since it is advantageous to the hardware to execute all instructions in the same number of steps, in RISC:

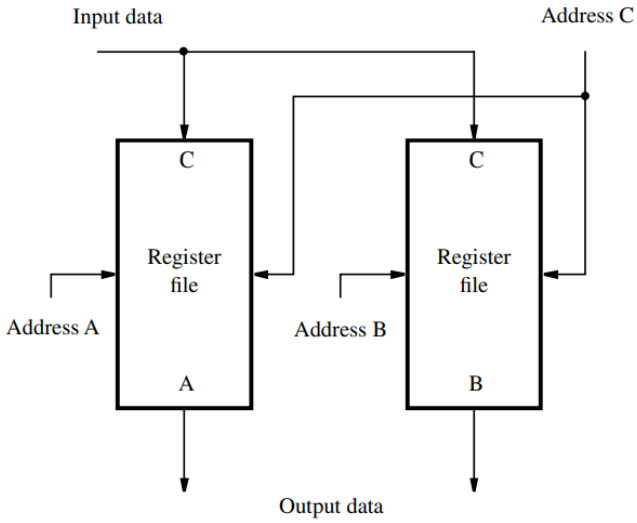
1. Same
2. Same
3. Same
4. No action
5. Same

Add R3, R4, #1000

1. Same
2. Decode the instruction and read register R4
3. Compute the sum [R4] + 1000
4. Same
5. Same

Store R6, X(R8) //Store has different order, SRC, DST

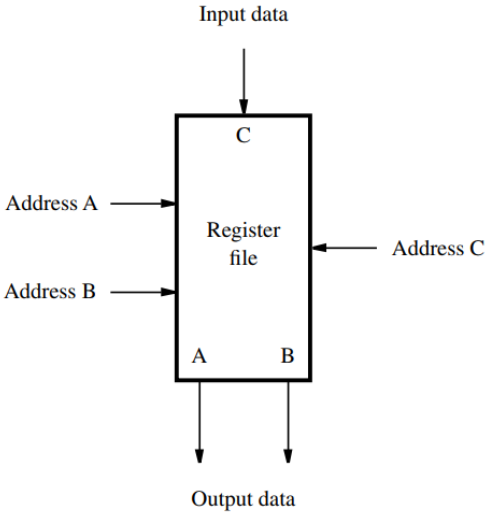
1. Fetch the instruction and increment the program counter
2. Decode the instruction and read registers R6 and R8
3. Compute the effectie address X + [R8]
4. Store the contents of register R6 into effective address
5. No action



(b) Two memory blocks

COMMON RISC INSTRUCTION 5 STAGES

1. **Fetch** an instruction and increment the program counter.
2. **Decode** the instruction and read registers from the register file.
3. **Execute** an ALU operation.
4. (if needed) Read or write memory data if the instruction involves a **memory** operand.
5. (if needed) Write the result into the destination **register**



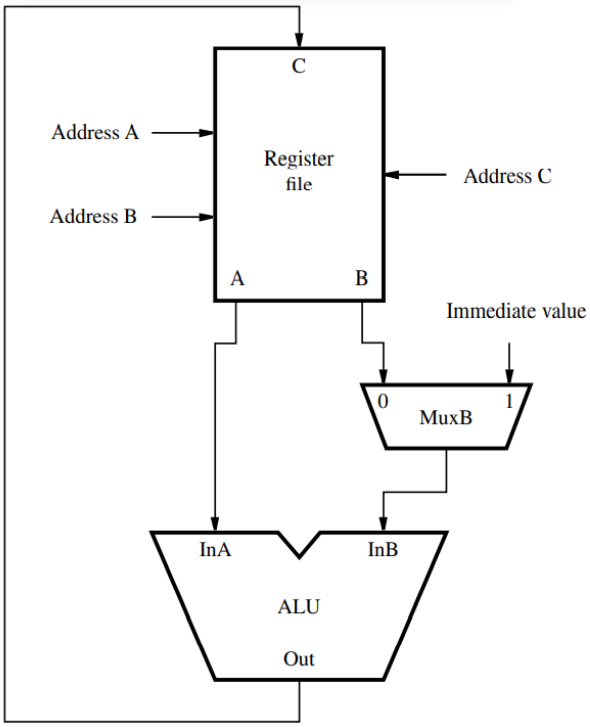
(a) Single memory block

In most RISC R0 = 0 and the default index registers value. When R0 is used as the index register, the effective address of the operand is the immediate value X. This is the Absolute addressing mode. Alternatively, if the offset X is set to zero, the effective address is the contents of the index register, Ri. This is the Indirect addressing mode. Thus, only one addressing mode, the Index mode, needs to be implemented, resulting in a significant simplification of the processor hardware.

REGISTER FILE

General purpose registers are implemented in the form of a register file that allows two registers to be read at the same time, its contents giving two separate outputs. The register files has 2 read addreses and 1 write address for a third register. The addreses inputs are connected to the IR field that specifies the DST. **ports:** inputs and outputs of any emory unit. **dual-ported:** memory unit that has two output ports.

ALU



The Arithmetic and Logic Unit is used to manipulate data:

1. perform arithmetic operations: addition, substraction
2. logic operations: such as AND, OR, XOR
3. It may be connected directly to the register file
   In CISC they'd have to go via the bus
4. The multiplexer selects either output B of the register file or the immediate value in the IR as the second ALU operand inB

INTER-STAGE REGISTERS

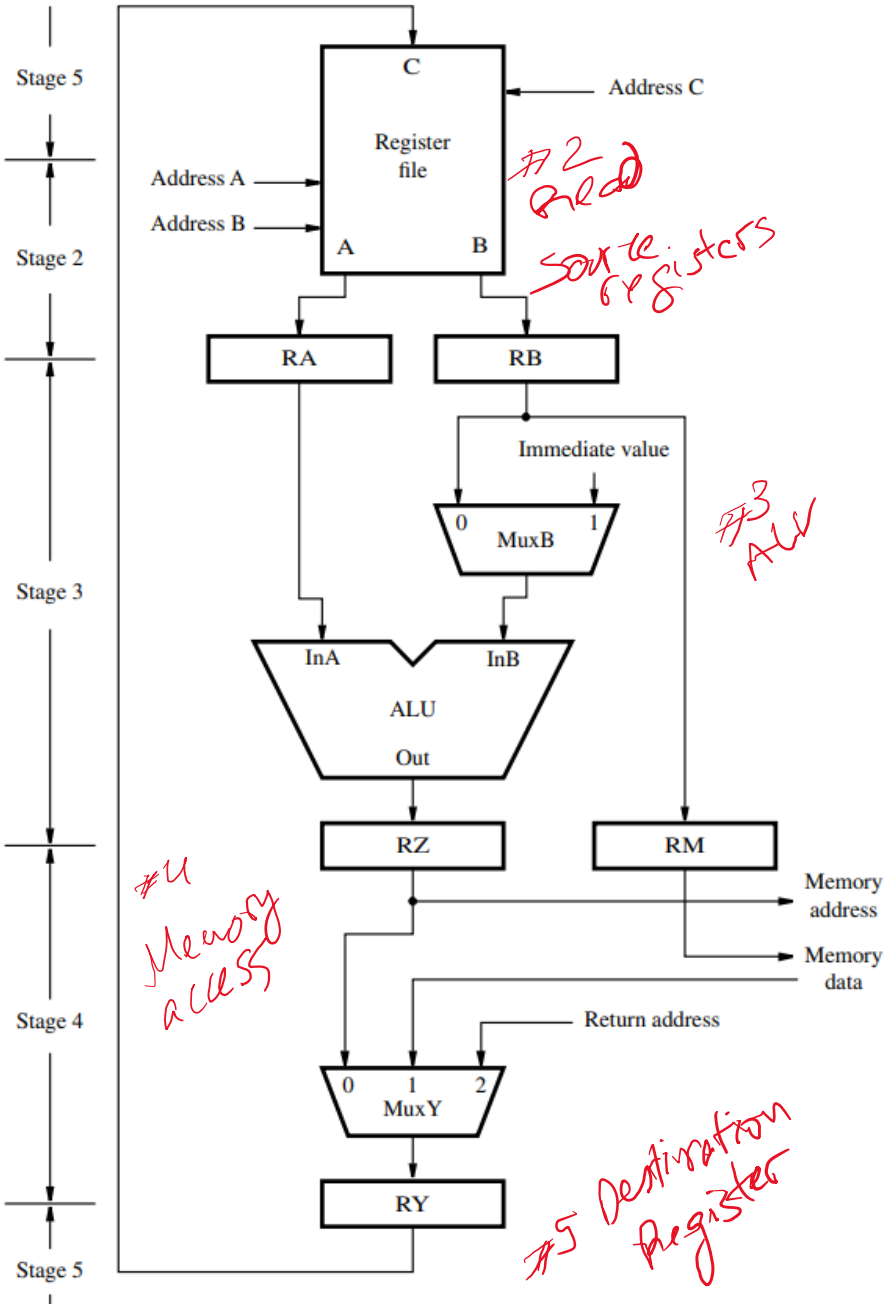
It is necessary to insert registers between stages. Inter-stage registers hold the results produced in one stage so that they can be used as inputs to the next stage during the next clock cycle. Which leads to the processor **datapath** structure on the right.

- Recall that for computational instructions, such as an Add instruction, no processing actions take place in step 4. During that step, multiplexer MuxYin selects register RZ to transfer the result of the computation to RY
- For Load and Store instructions, the **effective address** of the memory operand is computed by the ALU in step 3 and loaded into **register RZ**
- In the case of a **Load instruction**, the data read from the memory are selected by multiplexer MuxY and placed in **register RY**
- For a **Store instruction**, data are read from the register file, which is part of stage 2, and placed in **register RB**. Since memory access is done in stage 4, another **inter-stage register is needed to maintain correct data flow** in the multi-stage structure. Register **RM** is introduced for this purpose.

DATA PATH

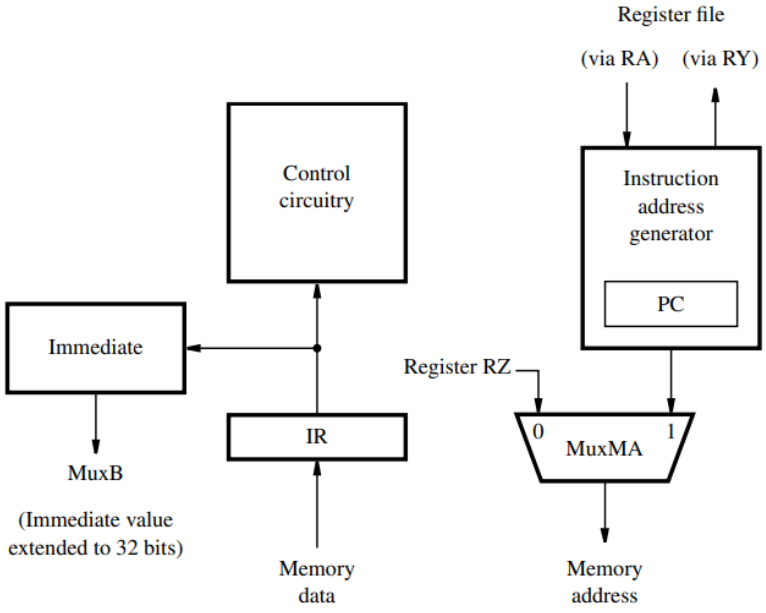
Since the instructions are based on two phases, fetch and execution, the hardware is also split in 2 corresponding sections. The fetching section also decodes the instruction and the control signals (between components), the other executes it: read operands, compute and store/load results.

Each of the 5 steps take 1 clock cycle.



- The general purpose register that holds the return addresses is called LINK.
- The general purpose register that holds interrupts addresses is called IRA
- The return address is produced by the instruction address generator

INSTRUCTION FETCH SECTION

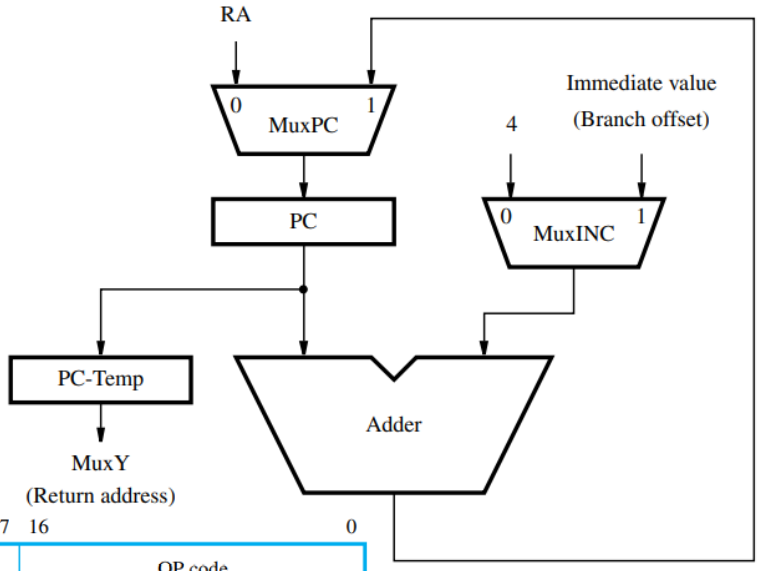


- The addresses used to access the memory come from the PC when fetching instructions and from register RZ in the datapath when accessing instruction operands.
- MuxMA selects one of this 2 sources.
- The instruction address generator updates the PC ater each instruction is fettched
- The instruction read from the memory is loaded into the IR, where it stays until its execution is completed and the next instruction is fetched.
- The contents of the IR are examined by the control circuitry to generate the signals needed to controll all the processor's hardware. They are alos used by the block labeled Immediate.
- A 16 bit IV can be extended to 32 bits, which will be used either as an ALU operand or as an index to compute the effective addres of an operand.
- The IV is sign extended or "padded" with zeros for arithmetic operations and logic instructions respectively.
- The IV also is used to compute the target address of branch instructions.

INSTRUCTION ADDRES GENERATOR

The Addres Generator Circuit on the right,

- Uses an adder to increment the PC value by 4 (4 byte difference in addresses)
- but it also computes the branch values.
- MuxINC selects constant 4 or branch
- MuxPC selects Adder result or RA
- PC-Temp holds tempraryily the PC contents due to interrupts or subroutine saves

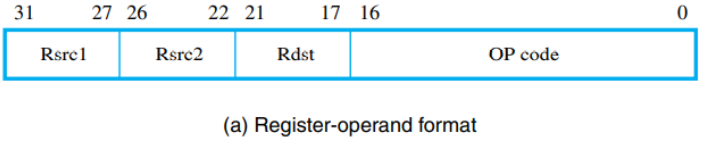


INSTRUCTION FETCH AND EXECUTION STEPS

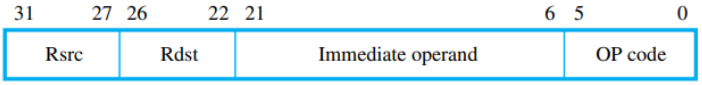
Add R3, R4, R5

Using the registers of **datapath** graph:

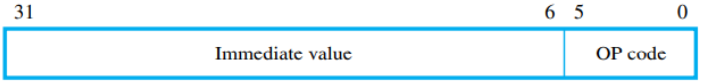
1. Memory address ← [PC],
   Read memory,
   IR ← Memory data,
   PC ← [PC] + 4
2. Decode instruction,
   RA ← [R4],
   RB ← [R5]
3. RZ ← [RA] + [RB]
4. RY ← [RZ]
5. R3 ← [RY]



(a) Register-operand format



(b) Immediate-operand format



(c) Call format

Instruction encoding.

Load R5, X(R7)

1. Memory address ← [PC], Read memory, IR ← Memory data, PC ← [PC] + 4
2. Decode instruction, RA ← [R7]
3. RZ ← [RA] + Immediate value X
4. Memory address ← [RZ], Read memory, RY ← Memory data
5. R5 ← [RY]

Store R6, X(R8)

1. Memory address ← [PC], Read memory, IR ← Memory data, PC ← [PC] + 4
2. Decode instruction, RA ← [R8], RB ← [R6]
3. RZ ← [RA] + Immediate value X, RM ← [RB]
4. Memory address ← [RZ], Memory data ← [RM], Write memory
5. No action

Note: *a memory Read or Write operation can be completed in one clock cycle when the data involved are available in the cache. When the operation requires access to the main memory, the processor must wait for that operation to be completed.*

Source register addresses are specified using the same bit positions in all instructions. The hardware reads the registers whose addresses are in these bit positions once the instruction is loaded into the IR

BRANCHING

The standard PC increment continues until a branch or subroutine call instruction loads a new address into the PC. Subroutine call instructions also save the return address. Interrupts from I/O devices and software interrupt are handled in a similar manner.

Branch instructions specify the branch target address **relative** (i.e. +5 or -3 lines back), A branch offeset given as an immediate value in the instruction is added to the current contents of the PC. The number of bits used for this offset is less than the word length of the computer. Therefore the range of addresses is limited. Subroutine call instructions reach a larger range of addresses as they have more available bits to specify the target addres and the RISC computers ahve jump and call instructions that use general-purpose registers to specify a full 32-bit address.

Step	Action
1	Memory address ← [PC], Read memory, IR ← Memory data, PC ← [PC] + 4
2	Decode instruction
3	PC ← [PC] + Branch offset
4	No action
5	No action

Sequence of actions needed to fetch and execute an unconditional branch instruction.

. In processors that do not use condition-code flags, the branch instruction specifies a compare-and-test operation that determines the branch condition. For example

Step	Action
1	Memory address ← [PC], Read memory, IR ← Memory data, PC ← [PC] + 4
2	Decode instruction, RA ← [R5], RB ← [R6]
3	Compare [RA] to [RB], If [RA] = [RB], then PC ← [PC] + Branch offset
4	No action
5	No action

Sequence of actions needed to fetch and execute the instruction:
Branch_if_[R5]=[R6] LOOP.

A simpler and faster comparator circuit can examine the contents of registers RA and RB and produce the required condition signals though. The comparator is usually inside the ALU and therefore not explicitly shown in graphs.

Subroutine calls and returns are implemented in a similar manner to branch instructions. The address of the subroutine may either be computed using an immediate value given in the instruction or it may be given in full in one of the general-purpose registers

Step	Action
1	Memory address ← [PC], Read memory, IR ← Memory data, PC ← [PC] + 4
2	Decode instruction, RA ← [R9]
3	PC-Temp ← [PC], PC ← [RA]
4	RY ← [PC-Temp]
5	Register LINK ← [RY]

Sequence of actions needed to fetch and execute the instruction:
Call_Register R9.

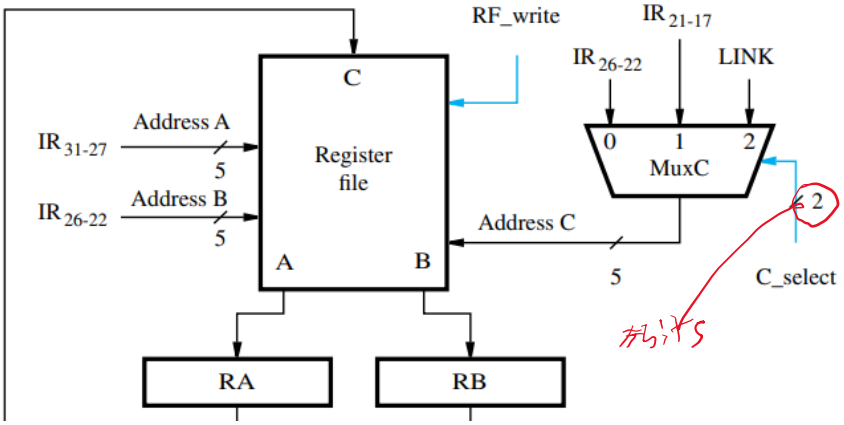
WAITING FOR MEMORY

The role of the processor-memory interface circuit is to control data transfers between the processor and the memory. Most of the times the instructions are found in the cache, which in that case the operation is compeleted in one clock cycle. When the information is not in the cache and has to eb fetched from the main memory several clock cycles may be needed. The **interface** circuit must inform the processor control circuitry to delay subsequent execution steps until the memory operation is completed. To do so there is a signal that needs to be checked, Memory Function Completed (MFC). When MFC is received, the processor proceeds to the next step. Step 1 of the execution sequence of any instruction involves fetching the instruction from the memory. Therefore, it must include a Wait for MFC command, as follows:

Memory address ← [PC], Read memory, Wait for MFC,
IR ← Memory data, PC ← [PC] + 4

The Wait for MFC command is also needed in step 4 of Load and Store instructions.

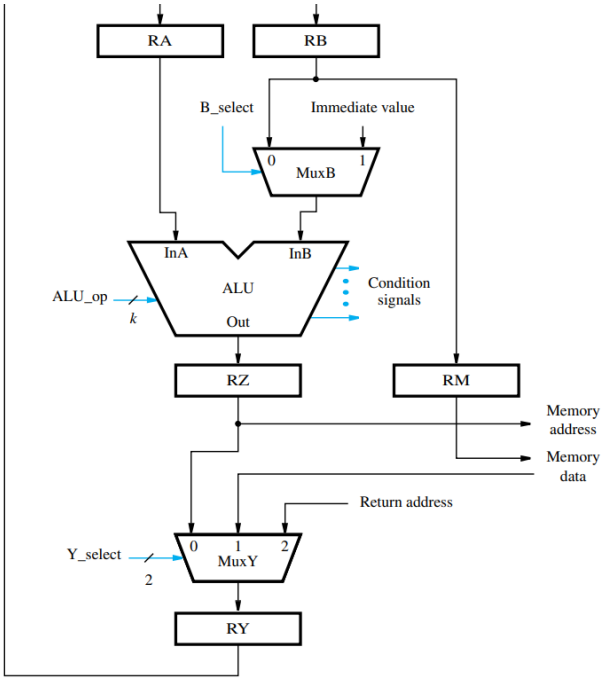
CONTROL SIGNALS



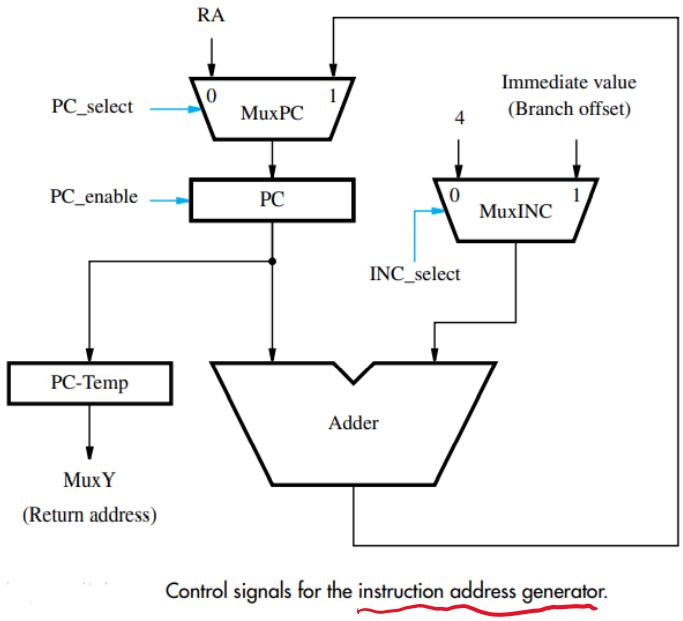
- The processor hardware components are governed by control signals, which determine which multiplexer input is selected, what operation is perfomed by the ALU, and whether to read or write a memory location or register.
- In each clock cycle (one for each step of the 5th step structure), intermediate results are stored in inter-stage registers RA, RB, RZ, RM, RY and PC-Temp, which are always enabled. The other registers must be enabled only when necessary and via control signals.

- The register file has three 5-bit address inputs, allowing access to 32 general-purpose registers. Two of these inputs, Address A and Address B, determine which registers are to be read. They are connected to fields IR31−27 and IR26−22 in the instruction register.
- The third address input, Address C, selects the destination register, into which the input data at port C are to be written.
- Multiplexer MuxC selects the source of that addres. 0 and 1 represent 2 possible IR slots for addresses.
- The third input of the multiplexer is the address of the link register used in subroutine linkage instructions
- New data are loaded into the selected register only when the control signal RF_write is asserted
- Multiplexers are controlled by signals that select which input data appear at the multiplexer's output

$$RF\_write = T5 \cdot (ALU + Load + Call)$$

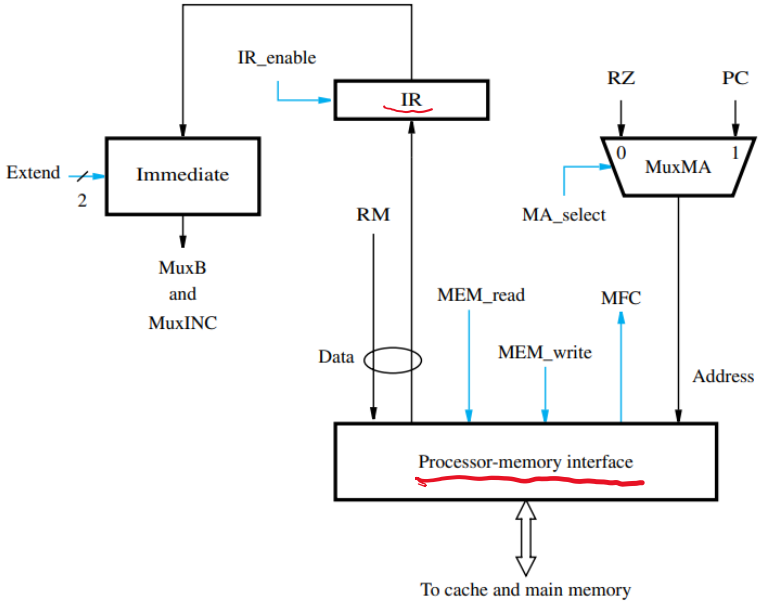


- Two signals, MEM_read and MEM_write are used to initiate a memory Read or a memory Write operation.
- When the requested operation has been completed, the interface asserts the MFC signal



Control signals for the instruction address generator.

- The operation performed by the ALU is determined by a k-bit control code.
- The comparator generates condition signals that indicate the result of the comparison. These signals are examined by the control circuitry during the execution of conditional branch instructions to determine whether the branch condition is true or false.



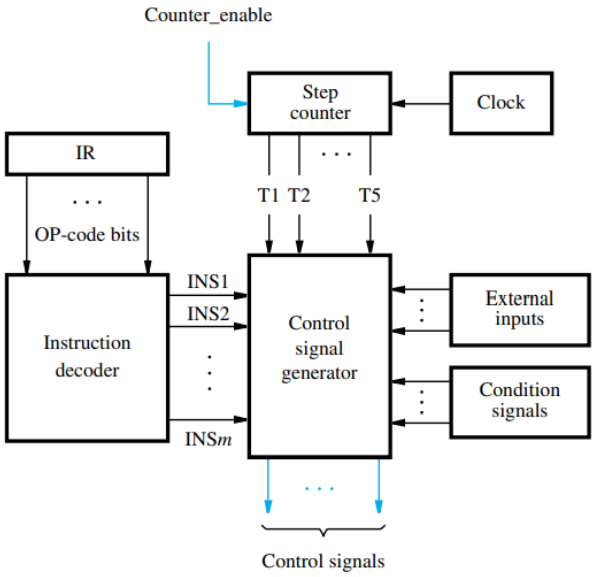
- The instruction register has a control signal, IR_enable, which enables a new instruction to be loaded into the register. When fetching it must be activated only after the MFC signal is asserted.
- The immediate value can be 1. sign extended 16-bit, zero-extended 16-bit, and a special 26-bit value. Hence its control signal Extend needs 2 bits.
- The INC_select signal selects the value to be added to the PC, either the constant 4 or the branch offset specified in the instruction.
- The PC_select signal selects either the updated address or the contents of register RA to be loaded into the PC when the PC_enable control signal is activated

B_select = Immediate

HARDWIRED CONTROL

There are 2 approaches to generate the control signals: hardwired control (RISC) and microprogrammed control (CISC). In hardwire control, the setting of the control signals depends on:

- Contents of the step counter
- Contents of the instruction register
- The result of a computation or a comparison operation
- External input signals, such as interrupt requests



- Instruction decoder interprets the OP-codeand he addressing mode information of the IR
- Instruction decoder sends the corresponding INS_i output to the control signal generator
- After each clock cycle the step counter signals either one of T1 to T5.

DEALING WITH MEMORY DELAY

- The timing signals T1 to T5 are asserted in sequence as the step counter is advanced. Most of the time, the step counter is incremented at the end of every clock cycle.
- When MEM_read or a MEM_write command is issued does not end until the MFC signal is asserted.
- To extend the duration of an execution step to more than one clock cycle, we need to disable the step counter:

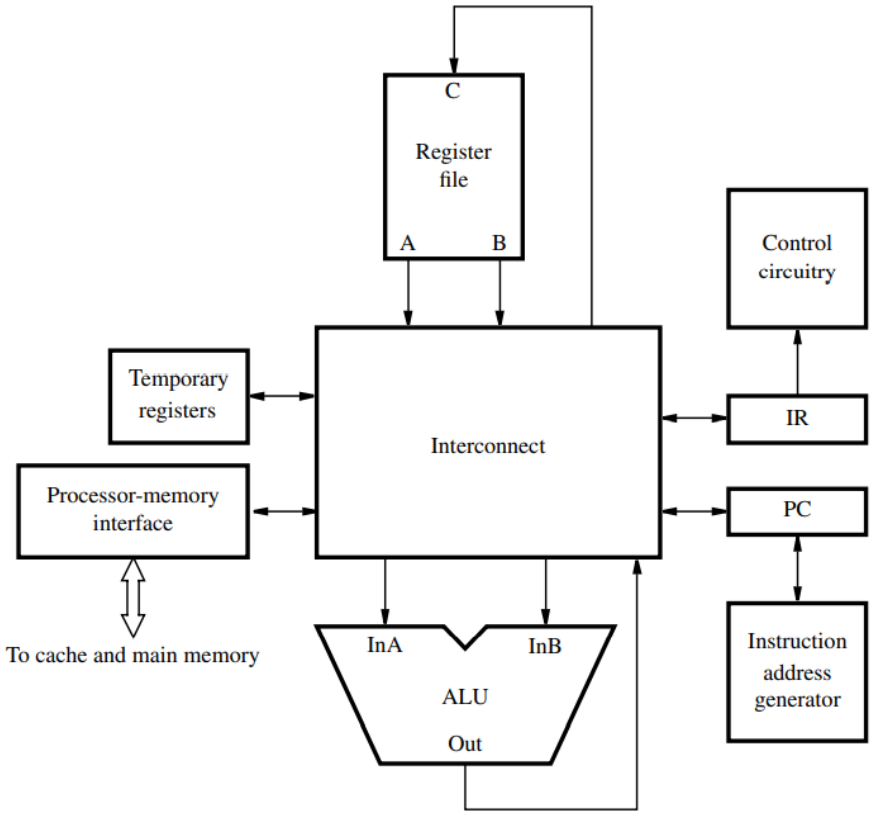
$$\text{Counter\_enable} = \overline{\text{WMFC}} + \text{MFC}$$

- A new value is loaded into the PC at the end of any clock cycle in which the PC_enable signal is activated.

The PC is incremented only once when an execution step is extended for more than one clock cycle. When fetching an instruction, the PC should be enabled only when MFC is received. It is also enabled in step 3 of instructions that cause branching (BR = instructions in branch group):

$$\text{PC\_enable} = \text{T1} \cdot \text{MFC} + \text{T3} \cdot \text{BR}$$

CISC STYLE (BOOK INTRO)



CISC-style instruction sets are more complex because they allow much greater flexibility in accessing instruction operands. Unlike RISC-style instruction sets, where only Load and Store instructions access data in the memory, CISC instructions can operate directly on memory operands. Also, they are not restricted to one word in length. Therefore, CISC-style instructions require a different organization of the processor hardware. Possible CISC processor suggested by the book on the left.

The **main difference** between this organization and the five-stage structure discussed earlier is that the **Interconnect block**, which provides interconnections among other blocks, does not prescribe any particular structure or pattern of data flow. It provides paths that make it possible to transfer data between any two components, as needed to implement instruction. **Inter stage registers are not needed but some registers are needed to hold intermediate results during instruction execution.**

CIS STYLE (LECTURES)

ISAs are needed to be standard so programs (and devices) are portable among different machines.

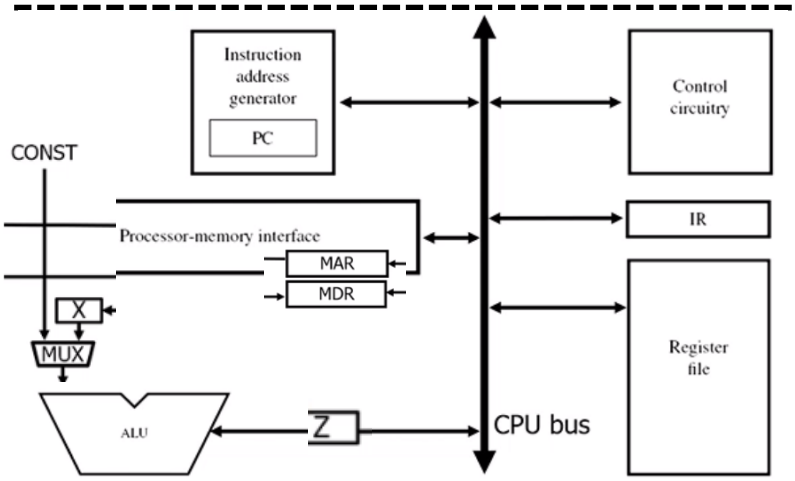
INSTRUCTION EXECUTION

- Divide and conquer: split the problem into multiple pieces solve them seperately and connect them.

RISC used 5 steps (page 4), CISC is flexible.

Since different parts of the hardware are allocated to different steps. You can start the fetching of the next instruction before the current one has been executed. The ALU is involved in load/store instructions in index addressing.

CIS CPU



Same components except for the CPU bus, the Register file is generally smaller (less registers) and the Control Circuitry includes bus behaviour.

CISC HAS 6 STAGES

Although it really depends on each computer. It generally will take more stages than RISC because there are more complex instructions that require more steps.

Parts involved in fetching an instruction:

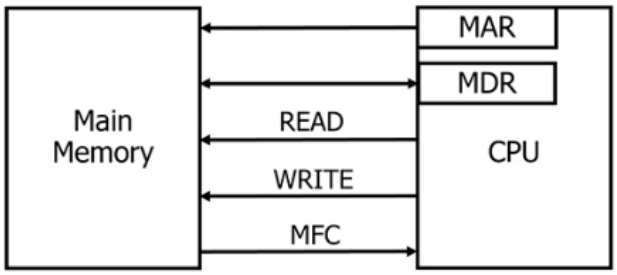
1. Lookup Memory Address Register
2. Increase/update program counter
3. Wait till Memory Data Register content arrives
4. Send Instruction to Instruction Register

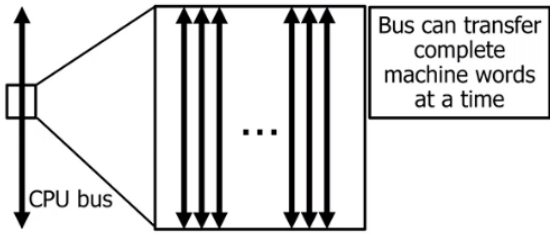
The bus connects all the different components.

ADD $16, R2, R4 // SRC, SRC, DST

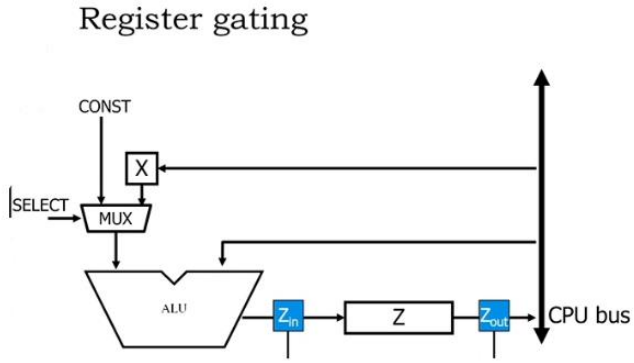
Stage 1. (already done, fetched the instruction and increased the program counter)
Stage 2. Decode instruction (Control Circuitry) (logic gates that look at the instruction and prepare things)
  i. Load constant/immediate value of 16 the ALU
  ii. Load Register in ALU with R2
Stage 3. (Skip) Reserved for fetching operands from memory/complex addressing
Stage 4. Execute and load the ALU result in Z register (Z is a necessary register to keep the results)
Stage 5. (Skip) Reserved for reading from memory
Stage 6. Move the register Z value to Register 4 accross the bus
Stage 7. (Skip) Reserved for writing to memory

INTERACTING WITH MAIN MEMORY AND THE BUS

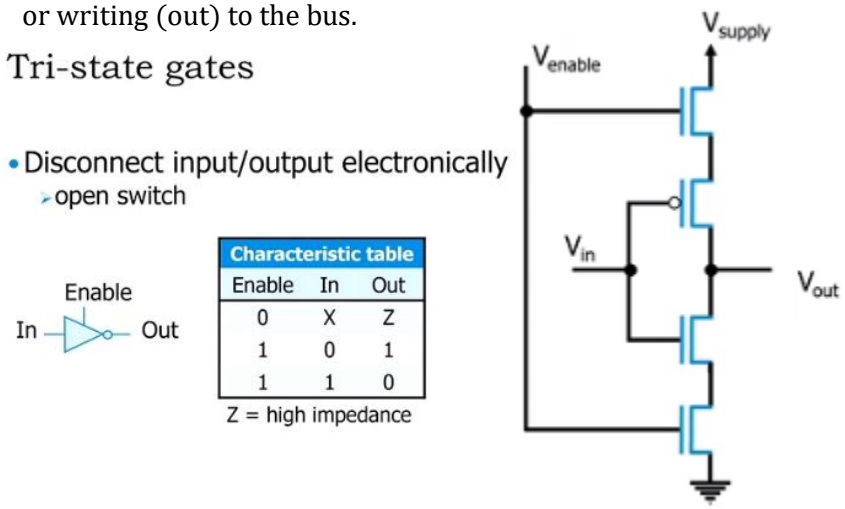




The control circuitry must disconnect all the other components not using the bus to avoid 1. data corruption and 2. interfering electronic signals (noise).

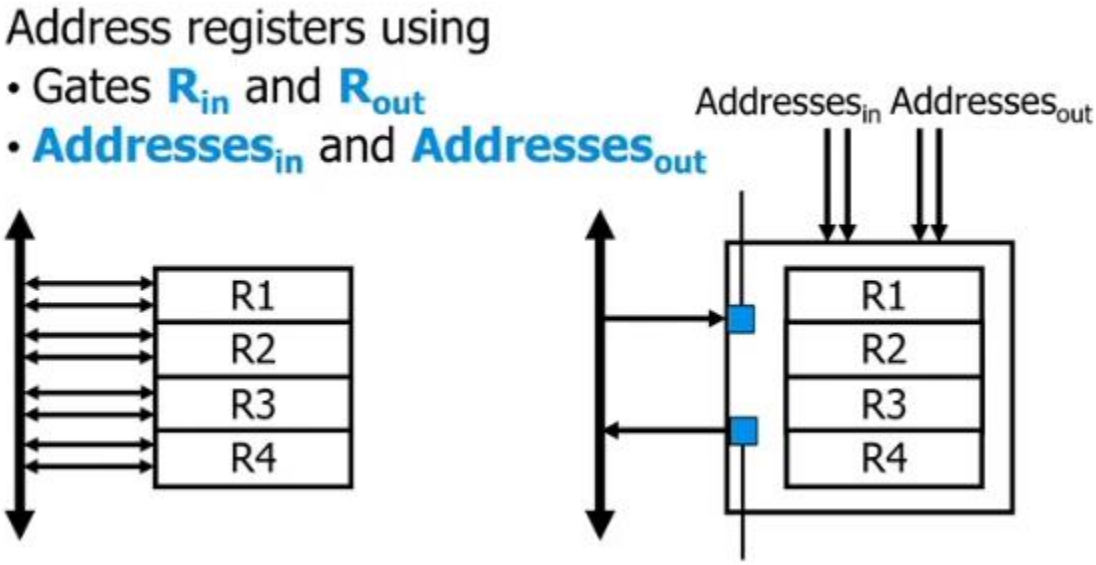


The tri-state gates/buffer wrapping register Z are the control circuit gates that refer to whether Z should be listening (in) or writing (out) to the bus.



Z = high resistance ≈ disconnecting from bus.
X I assume is the last value on the Bus.

REGISTER FILE



You do it like this instead of adding an Rin Rout n times for each Register. Otherwise the mictro instruction length would be too long and the suggested architecture is actually pretty efficient for moving contents between registers.

STORE R2, (R1) // SRC, DST indirect addressing mode, R1 contains the value of the memory address

Stage 1. Fetch

a. PCout
b. MARin
c. Read
d. WMFC (till here one cycle)
e. MDRout
f. IRin (another cycle?)

Stage 2. Decode instruction

g. R1out
h. MARin (one cycle)
i. R2out
j. MDRin (another cycle)

Stage 7. Write to memory

k. Write
l. WMFC (last cycle)

In CISC we can skip stages if we dont need them.

To improve CPU performance you can duplicate components (such as the bus, which would allow to parallel transfers), (the ALU or the register files could also be duplicated) and/or pipeline parts of instructions.

CONTROL CIRCUITRY

Manipulates the control lines (tri-state gates that control the in/out behaviour of the registers connected to the bus)

STORE R2, (R1)

1. PC_out, MAR_in, read, WMFC
2. MDR_out, IR_in
3. R1_out, MAR_in
4. R2_out, MDR_in, write, WMFC

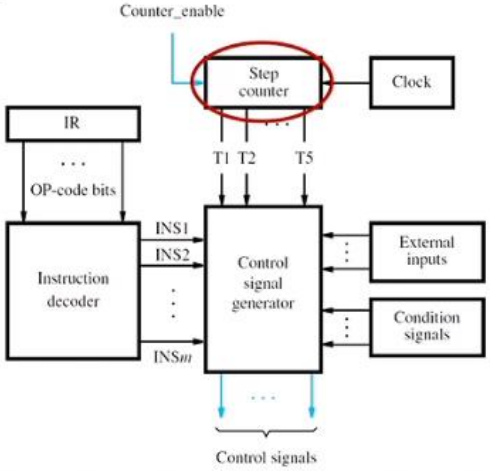
Conceptual task of control circuitry

"Signal table" to look up all instruction stages

	PC_{out}	PC_{in}	$R1_{in}$	$R1_{out}$	MAR_{in}	MDR_{out}	IR_{in}	...	WMFC
⇨1	1	0	0	0	1	0	0		1
2	0	0	0	0	0	1	1		0
3	0	0	0	1	1	0	0		0
4	...								
5	...								
6	...								
7	...								

Fixed format per opcode in RISC (every instruction has virtually the same operands)
and for CISC the look up table regards the different possible addressing modes.

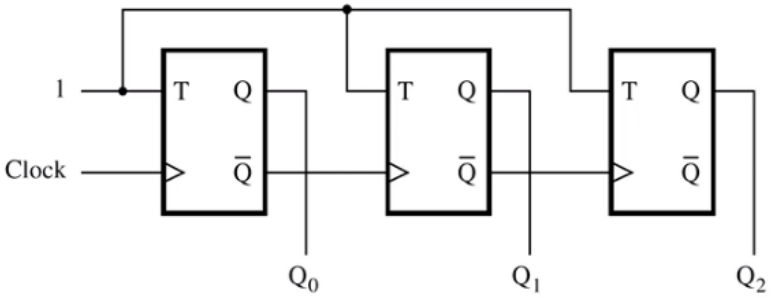
Step counter



1. First fetch the next instruction.
2. Then decode what needs to be done
3. Then execute the thing
4. Then write the results back at the right place

The step counter regards each of those steps above

T = step (tick). Counter below

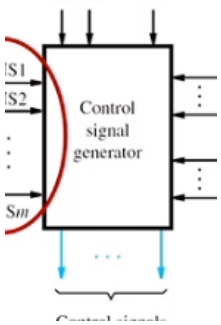


Control signals depend on...

- Instruction code
  - What kind of instruction are we executing?
- Control step counter
  - Which stage of the instruction are we currently in?
- Flags & external signals
  - Are there signals that change program behavior?

An example of a flag that changes the program behaviour is the WMFC and status flags in register for comparistion and branches, zero flag.

HARDWIRED CONTROL



Dedicated circuitry that makes up the box in the middle. which takes inputs from the step counter, instruction register decoder, condition signals and external inputs.

The counter is controlled by the MFC and WMFC signals:

$$\text{Counter\_enable} = \overline{\text{WMFC}} + \text{MFC}$$

(ALU or Load or Call) were executed

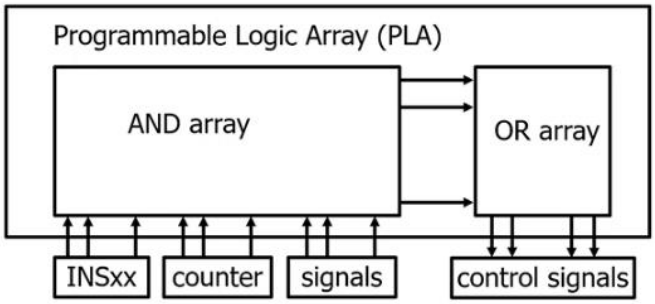
To write anything you always need to be on tick T5 and

RF_write = T5 · (ALU + Load + Call)

Calling an address requires to write/load the return addres on LINK register and occasionall mess with the stack pointer so depending on how you implement call you have to write 1 or 2 registers

Control Signal Generator

By PLA (or ASIC)



RF_write = T5 · (ALU + Load + Call) = T5 · ALU + T5 · Load + T5 · Call

Programmable Logic Array on the left. You defnie which 2 signals need to be "ANDed"

The end array generates the ANDs. AND the OR array takes all elements of the AND to make a sum of products

Hardwired control

(dis)advantages

Implemented in hardware

- so it's fast!
- so it's inflexible!

BPLA = Programmable Logic Array = Hardware Approach to generate control signals

MICRO PROGRAMMED CONTROL

A little processor instide the processor. So. in practice, an opcode is nothing but a menonic that stands for a set of micro instructions, which enable/disable the tri-state gates that wrap the components interacting with the bus.

Microprogram 'control store'

Signal table stored in ROM

micro program memory

	PC_{out}	PC_{in}	$R1_{in}$	$R1_{out}$	MAR_{in}	MDR_{out}	IR_{in}	...	WMFC
1	1	0	0	0	1	0	0		1
2	0	0	0	0	0	1	1		0
3	0	0	0	1	1	0	0		0
4	...								
5	...								
6	...								
7	...								

control word

- Increased flexibility (branching)

The rows represent the steps/ticks of an opcode. In CISC that would be 5 (fetch, decode, alu, r/w memory, w register) whereas in CISC 7 (fetch, decode, fetch complex addressing operands, alu, r memory, move registers, w memory).

Microinstructions

Like instructions, but different

Instruction

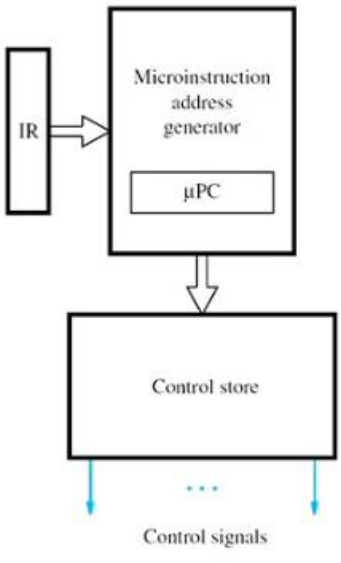
MOV	R1	A
MOV	R1	R2

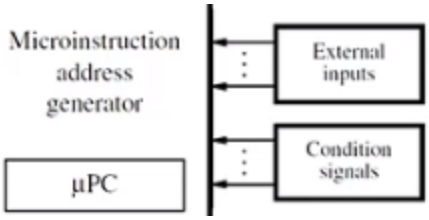
Microinstruction

PC_{out}	PC_{in}	IR_{in}	...	END
PC_{out}	PC_{in}	IR_{in}	...	END
PC_{out}	PC_{in}	IR_{in}	...	END

After fetching we can branch to a standard address of sum, substr, call etc. operations.
Micro instructions in CISC, unlike the instructions, are of fixed length.

A CPU inside a CPU ->
Compared to hardwire, this uses branching instead of sequencing.
Comapred to a full CPU, there's no ALU nor registers





Don't forget the external inputs and conditions signals.

END bit: It is generally cheaper to have 1 bit in each microinstruction than to have a full word dedicated to a jump/call function to end the program.
1 bit * 5 or 7 ticks is less than 1 * 23ish bit word length.

Microroutine example

ADD R1, R2

Address	Micro-instruction	
0	PC_out, MAR_in, Read, WMFC	Fetch Instruction
1	MDR_out, IR_in	
2	Branch to starting address routine (here, 42)	**Decode**
42	RF_read, RF_addr_out = 1, X_in	Exec
43	RF_read, RF_addr_out = 2, MUX_x, F_alu = "ADD", Z_in	
44	Z_out, RF_write, RF_addr_in = 2, **End**	

Remember that jumps are relative not absolute so jump 10 means jump to instruction +10.

BNZ PC+OFFSET

If Z=0 then go to address 0

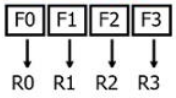
Address	Micro-instruction	
0	PC_out, MAR_in, Read, WMFC	Fetch Instruction
1	MDR_out, IR_in	
2	Branch to starting address routine (here, **103**)	**Decode**
103	PC_out, X_in, if Z=0 then **goto address 0**	← Test Z bit
104	Offset-field-of-IR_out, MUX_x, F_alu = "ADD", Z_in	
105	Z_out, PC_in, **End**	← New PC address

The number of control lines doesn't change if the registers are 64 bits or 32 bits, the number of control lines regard the components connected to the bus.

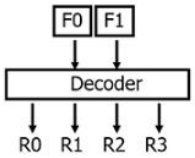
Control-store organization

Micro-instruction layout

1. No / little encoding: **horizontal organization**
   1. Large words
   2. Less decoding
   3. Faster



2. Much encoding: **vertical organization**
   1. Small words
   2. More decoding
   3. Slower



3. Mixed encoding

Control-store organization

Mixed encoding

Horizontal organization

Vertical organization

micro-instruction	PC_{out}	IR_{in}	MAR_{in}	MDR_{out}	REG_{in} addr
1	1	0	1	0	00
2	0	1	0	1	00
3	0	0	0	0	10
..					
..					

Mix & match

Encoding type per field

F1	F2	F3	F4	F5	F6	F7	F8

Field 1 (4 bits): Register address_in
Field 2 (4 bits): Register address_out
Field 3 (2 bits): RF_in/out
Field 4 (4 bits): Function ALU
Field 5 (2 bit): Read/Write/Nop
Field 6 (1 bit) : Carry-in ALU
Field 7 (1 bit) : WMFC
Field 8 (1 bit) : End

Microprogrammed control

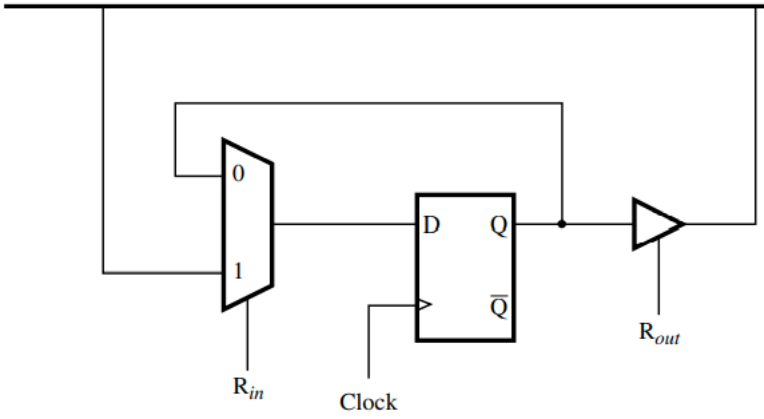
(dis)advantages

Implemented as a 'software' program

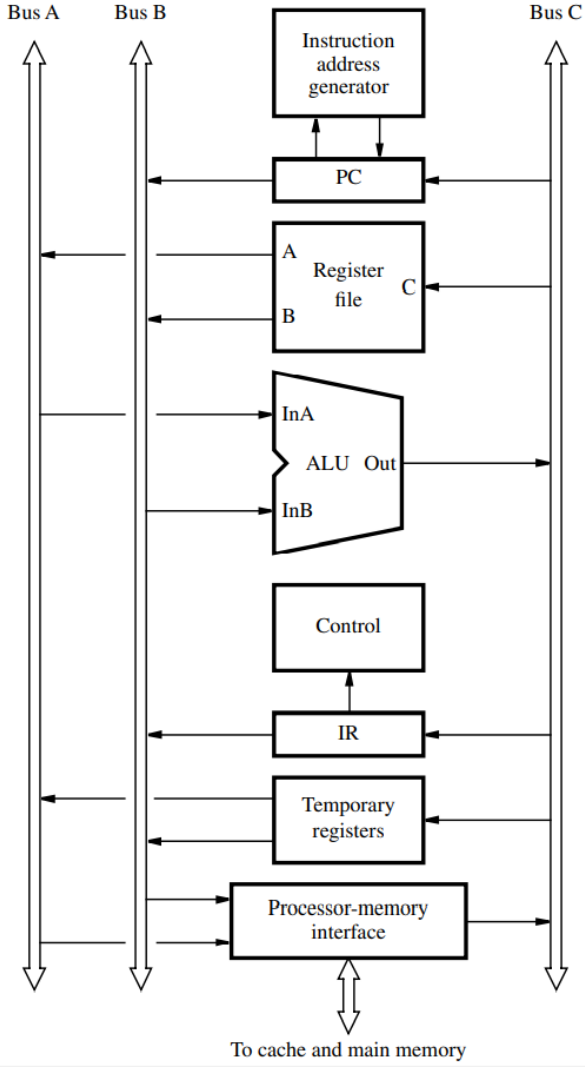
- so it's flexible!
- so it's slower!
- so it's more complex to build in hardware!

CIS STYLE (BOOK WRAP UP)

A bus consists of a set of lines to which several devices may be connected, enabling data to be transferred from any one device to any other. A logic gate that sends a signal over a bus line is called a **bus driver,** which can be only 1. A **flip-flop**s make the Rin and Rout registers.



Input and output gating for one register bit.



Alternative bus architecture by using 3 buses.

Step	Action
1	Memory address ← [PC], Read memory, Wait for MFC, IR ← Memory data, PC ← [PC] + 4
2	Decode instruction
3	R5 ← [R5] + [R6]

Sequence of actions needed to fetch and execute the instruction: Add R5, R6.

Step	Action
1	Memory address ← [PC], Read memory, Wait for MFC, IR ← Memory data, PC ← [PC] + 4
2	Decode instruction
3	Memory address ← [PC], Read memory, Wait for MFC, Temp1 ← Memory data, PC ← [PC] + 4
4	Temp2 ← [Temp1] + [R7]
5	Memory address ← [Temp2], Read memory, Wait for MFC, Temp1 ← Memory data
6	Temp1 ← [Temp1] AND [R9]
7	Memory address ← [Temp2], Memory data ← [Temp1], Write memory, Wait for MFC

Sequence of actions needed to fetch and execute the instruction: And X(R7), R9.

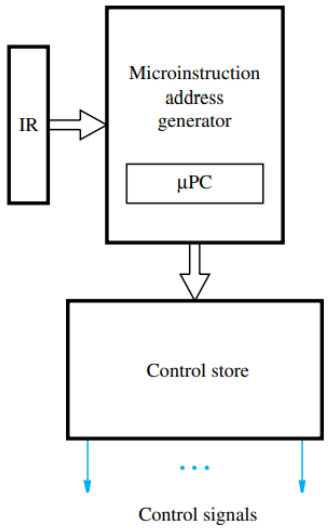
Step	Action
1	Memory address ← [PC], Read memory, Wait for MFC, IR ← Memory data, PC ← [PC] + 4
2	Decode instruction
3	Memory address ← [PC], Read memory, Wait for MFC, Temp1 ← Memory data, PC ← [PC] + 4
4	Temp2 ← [Temp1] + [R7]
5	Memory address ← [Temp2], Read memory, Wait for MFC, Temp1 ← Memory data
6	Temp1 ← [Temp1] AND [R9]
7	Memory address ← [Temp2], Memory data ← [Temp1], Write memory, Wait for MFC

Sequence of actions needed to fetch and execute the instruction: And X(R7), R9.

MICROPROGRAMMED CONTROL

Control signals are generated for each execution step based on the instruction in the IR and they control what happens on the bus. To do so there is a microprogram stored on the processor chip in a small and fast memory called the microprogram memory or the control store. . Let each control signal be represented by a bit in an n-bit word, which is often referred to as a control word or a microinstruction. (bit such us the one of a component either reading or listening from the bus, PC enable, WMFC, etc).

The sequence of microinstructions corresponding to a given machine instruction constitutes the microroutine that implements that instruction. The microprogrammed control unit is shown below.



Furthermore, the address generator uses a microprogram counter.

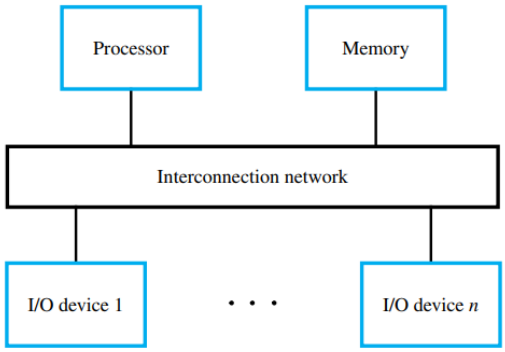
Microprogrammed control is simple to implement and provides considerable flexibility in controlling the execution of machine instructions. But, it is slower than hardwired control.

Modern processors have a multi-stage organization because this is a structure that is wellsuited to pipelined operation.

Solved problems at page 209 of the book

CHAPTER 3 AND 7: INPUT/OUTPUT

MEMORY-MAPPED I/O



A computer system.

- The idea of using addresses to access various locations in the memory and registers can be extended to accessing various devices.
- Each I/O device must appear to the processor as consisting of some addressable locations, just like the memory.
- **memory-mapped I/O:** Some addresses in the address space of the processor are assigned to these I/O locations, rather than to the main memory.
- These locations are usually implemented as bit storage circuits (flip-flops) organized in the form of "I/O registers"

Load R2, DATAIN //DST, SRC: reads the data from the DATAIN register and loads them into processor register R2.

Store R2, DATAIN //SRC, DST: sends the contents of register R2 to location DATAOUT, which is a I/O register.

I/O DEVICE INTERFACE

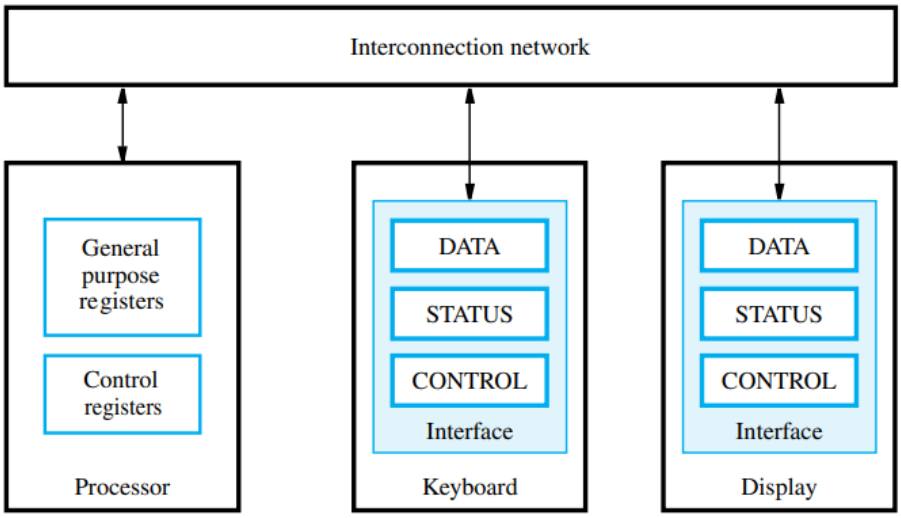


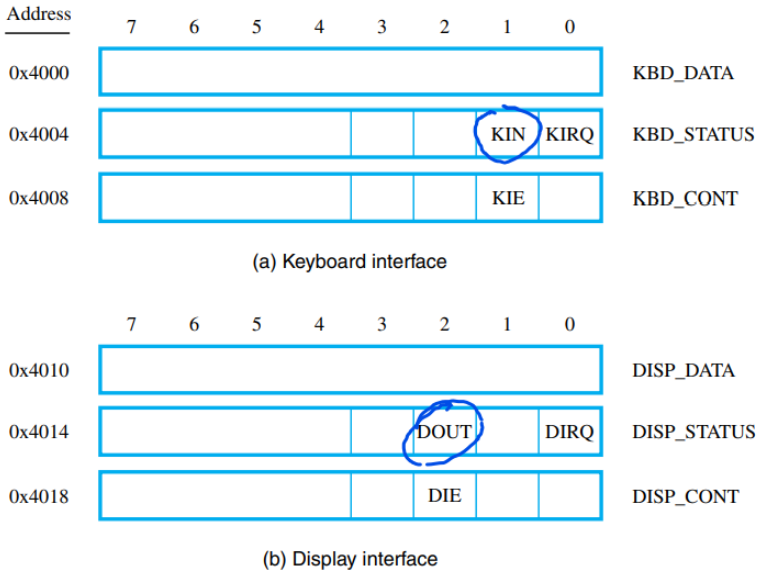
Figure 3.2 The connection for processor, keyboard, and display.

- An I/O device is connected to the interconnection network by using a circuit, called the **device interface**.
- The interface includes some registers among them **data, status, and control registers** whcih are accessed by program instructions **as if they were memory locations**.

PROGRAM-CONTROLLED I/O

- Consider a task that **reads** characters typed on a keyboard, **stores** these data in the memory, and **displays** the same characters on a display screen. A simple way of implementing this task is to write a program that performs all functions needed to realize the desired action.

- Responeses from the keyboard must be done in a timely manner.
- The rate of data transfer from the keyboard to a computer is limited by the typing speed of the user, which is unlikely to exceed a few characters per second
- The rate of output transfers from the computer to the display is determined by the rate at which characters can be transmitted to and displayed on the display device, typically several thousand characters per second.
- The difference in speed between I/O devices creates the need to syncronize data transfer between them



Registers in the keyboard and display interfaces.

- **signaling protocol:** On **output**, the processor **sends** the **first** character and then **waits** for a signal from the display that the **next** character can be sent. **Then sends** the **second** character, and so on.
- On **input:** The **processor waits** for a **signal** indicating that a key has been pressed and that a binary code that represents the corresponding character is **available** in an **I/O register** associated with the keyboard. **Then** the processor proceeds to **read** that value.
- **Polling**: The **processor reads a status flag** (such as KIN = key has been pressed) which is part of an 8 bit status register (KBD_STATUS)
- If the registers in I/O interfaces are to be accessed as if they are memory locations, **each register must be assigned a specific address** that will be recognized by the interface circuit.
- All addresses should be **word-aligned.** This makes the I/O registers accessible in a program executed by the processor

Assume that the initial state of KIN is 0 and the initial state of DOUT is 1, which is normally performed by the device control circuits when power is turned on

- Read the character pressed to register 5: READWAIT

```
Read the KIN flag
Branch to READWAIT if KIN=0
Transfer data from KBD_DATA to R5
```

- To display the character from register 5: WRITEWAIT

```
Read the DOUT flag
Branch to WRITEWAIT if DOUT = 0
Transfer data from R5 to DISP_DATA
```

The wait loop is executed repeatedly until the status flag DOUT is set to 1 by the display when it is free to receive a character. Then, the character from R5 is transferred to DISP_DATA to be displayed, which also clears DOUT to 0.

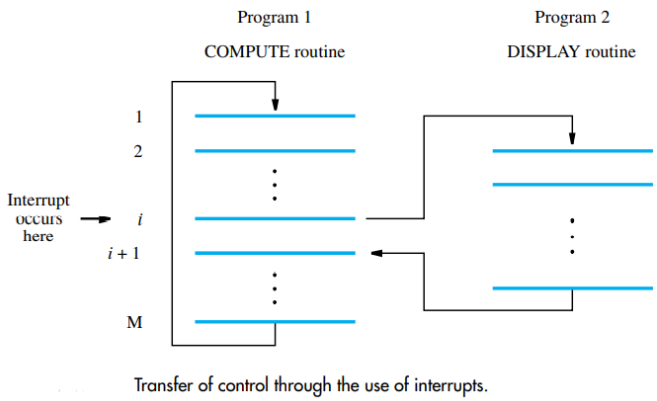
- In computers that use memory-mapped I/O you could implement it as follows (RISC-style)

```
READWAIT:   LoadByte        R4, KBD_STATUS   WRITEWAIT:   LoadByte        R4, DISP_STATUS
            And             R4, R4, #2                    And             R4, R4, #4
            Branch_if_[R4]=0 READWAIT                     Branch_if_[R4]=0 WRITEWAIT
            LoadByte        R5, KBD_DATA                  StoreByte       R5, DISP_DATA
```

Recall that KIN is b1 ($10_{base\ 2} = 2$)and DOUT is b2 ($100_{base2} = 4$)

INTERRUPTS

When the processor is polling in a loop like in the example above it cannot perform other tasks. Instead, we can arrange for the I/O device to alert the processor when it becomes ready by sending a hardware signal called an interrupt request to the processor. An example solution is to sperate the computation and I/O routines, and let the processor compute most of the time and occasionally jump to the I/O routines. Such jump is the interrupt.



Transfer of control through the use of interrupts.

- The routine executed in response to an interrupt request is called the **interrupt-service routine,** which are similar to subroutine calls.
- When the interrupt occurs, the PC saves the current PC value in temp register LINK or IPS, then PC is updated with the interrupt adddress, the procesor executes it, then resumes the previous routine.
- As part of handling interrupts, the **special control signal "intterupt acknolwedge" from the processor informs the device that its request has been recognized** so that it may remove its interrupt-request signal. It is sent to the device through the interconnection network.
- **Before starting execution of the interruptservice routine, status information and contents of processor registers that may be altered in unanticipated ways during the execution of that routine must be saved.**

- **Return-from-interrupt instruction:** The **saved information is restored before execution of the interrupted program is resumed**. In this way, the original program can continue execution without being affected (except delay) by the interruption.
- **interrupt latency:** delay from the register savings before executing interrupt. Typically, the processor saves only the contents of the program counter and the processor status register
- Some computers provide two types of interrupts. One saves all register contents, and the other does not.
- **shadow registers:** a different set of registers can be used by the interrupt-service routine, thus eliminating the need to save and restore registers.
- **real-time processing:** The concept of interrupts used in operating systems and in control applications where processing of certain routines must be accurately timed relative to external events.

ENABLING AND DISABLING INTERRUPTS

It must still be within the programmers power to control whether interrupts are enabled or not. It should be possible to enable/disable interrupts both at processors and I/O device ends. To do so we use control bits in registers that can be accessed by program instructions.

- **status register (PS):** processor registor that contains information about its current state or operation, when 1, interrupt is allowed, when 2 interrupts are ignored.
- The I/O devices also have a control register that contain the information about how themselves should be operated.
- When a device activates the interrupt-request signal, it keeps this signal activated until it learns that the processor has accepted its request.
- It is essential to ensure that this active request signal does not lead to successive interruptions, causing the system to enter an infinite loop from which it cannot recover.
- A good choice is to have the processor automatically disable interrupts before starting the execution of the interrupt-service routine.
- The processor saves the contents of the program counter and the processor status register.
- After saving the contents of the PS register, with the IE bit equal to 1, the processor clears the IE bit in the PS register, thus disabling further interrupts.
- Then interrupt-service routine starts, followed by the Return-from-interrupt instruction
- Which restores the contents of the PS registe, sets the IE bit back to 1, and therefore interrupts are again enabled (but not looped, it is still up to the processor to decide when to pick up the request).

1. The device raises an interrupt request.
2. The processor interrupts the program currently being executed and saves the contents of the PC and PS registers.
3. Interrupts are disabled by clearing the IE bit in the PS to 0.
4. The action requested by the interrupt is performed by the interrupt-service routine, during which time the device is informed that its request has been recognized, and in response, it deactivates the interrupt-request signal
5. Upon completion of the interrupt-service routine, the saved contents of the PC and PS registers are restored (enabling interrupts by setting the IE bit to 1), and execution of the interrupted program is resumed.

HANDLING MULTIPLE DEVICES

Multiple devices, that are operationally independent, sending interrupt requests are not syncrhonised. To fix this:

- When an interrupt request is received it is necessary to identify the particular device that raised the request
- if two devices raise interrupt requests at the same time, it must be possible to break the tie and select one of the two requests for service (and then execute the other).
- The information needed to determine whether a device is requesting an interrupt is available in its status register. When the device raises an interrupt request, it sets to 1 a bit in its status register, which we will call the **IRQ bit**.
- The simplest way to identify the interrupting device is to have the interrupt-service routine **poll all I/O devices in the system**

- The first device encountered with its IRQ bit set to 1 is the device that should be serviced. (first-in-first-out)

VECTORED INTERRUPTS

The main disadvantage of the previous last step is the time spent interrogating the IRQ bits of devices that may not be requesting any service.

- **vectored interrupts:** interrupt-handling schemes where teh device identifies iteself t othe processor rather than the processor polling for devices.
- A device requesting an interrupt can identify itself if it has its own interrupt-request signal, or if it can send a special code to the processor through the interconnection network
- **interrupt-vector table:** permanently allocated area in the memory to hold the addresses of interrupt-service routines, these addresses are alos called **interrupt vectors**.

INTERRUPT NESTING

- Generally, interrupts should be disabled during the execution of an interrupt-service routine, to ensure that a request from one device will not cause more than one interruption (aka. interrupt himself).
- Hower sometimes it is desired that high priority devices may be able to interrupt lower prioiritiy devices.
- A processor priority level can be assigned, which can be encoded in a few bits of the processor status register
- For each nested interrupt service routine the stack must save the program counter and the status register, which has to be done before the interrupt-service routines enables nesting.

CONTROLLING I/O DEVICE BEHAVIOUR

- **control register:** register **in the device** interface that hodls information needed to control the device
- The control register is accessed as an addressable location, just like the data and status registers. In a 32-bit processor, the control registers are 32 bits long.
- **interrupt-enable**: bit in the control register of the device that stores whether the processor will recognise it
- ***IRQ**: bit that is set to 1 if an interrupt request has been rised but not yet serviced

PROCESSOR CONTROL REGISTERS

- To deal with interrupts, besides the status register (PS) with the intterupt-enable bit (IE), other registers and bits shall be used. The IPS saves the content of PS when an interrupt request is received and accepted.
- After the interrupt-service routine, the previous state of the processor is restored from IPS to PS. If nested interrupts are used then IPS must use the stack.
- **IENABLE:** allows the processor to slectively respond to individual I/O devices, where a bit is assigned for each device.
- **IPENDING:** register that indicates the active interrupt requests (usefol for when multiple devices make requests at the same time).
- control registers cannot be accessed in the same way as the general-purpose registers. They cannot be accessed by arithmetic and logic instructions, nor by Load and Store in the same enconding format.
- Therefore they have their own dedicated special instructions:

MoveControl R2, PS //DST, SRC

CISC INTERRUPTS

- CISC can test status bits of I/O registers directly."TestBit" instruction is used to test the status flag.
- SetBit and ClearBit will make it 1 and 0 respectively.

EXCEPTIONS

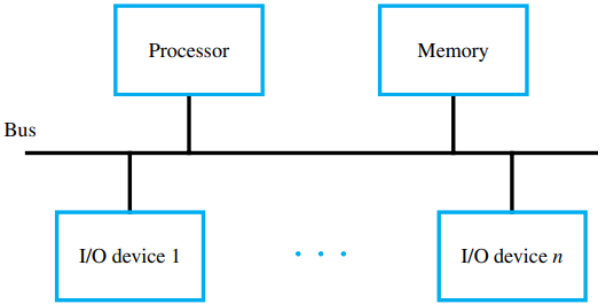
- An interrupt is an event that causes the execution of one program to be suspended and the execution of another program to begin
- **exception:** refers to any event that causes an interruption, which is not limited to just I/O interrupts
    - **recovery from errors:** If an error occurs (in the hardware), the control hardware detects it and informs the processor by raising an interrupt. For example, The OP-code field of an instruction may not correspond to any legal instruction, or an arithmetic instruction may attempt a division by zero.

- when an interrupt is caused by an error associated with the current instruction, that instruction cannot usually be completed, and the processor begins exception processing immediately

- **debuggings:** A debugger usex exceptions to allow trace mode and breakpoints features, which interrupt the instructions at specific points.
- **operating system:** the OS software may use exceptions to communicate/control with the execution of user programs. It also uses hardware interrupts to perform I/O operations.

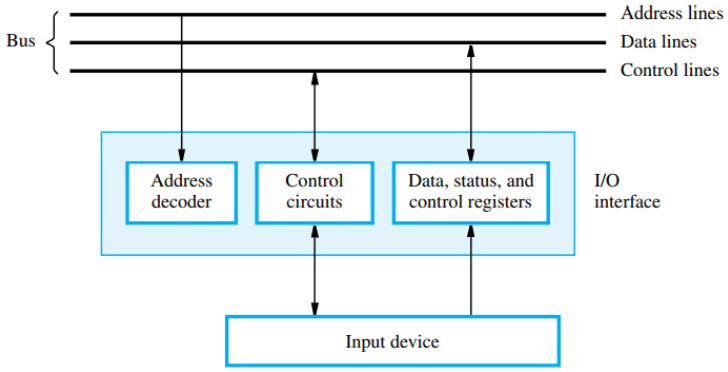
BUS STRUCTURE

The bus is the interconnection network that is used to transfer data among the processor, memory, and I/O devices.



A single-bus structure.

- Only one source/destination pair of units can use this bus to transfer data at any one time.
- The bus consists of three sets of lines used to carry: address, data and control signals
- Each I/O device is assigned a unique set of addresses for the registers in its interface.
- When the processor places a particular address on the address lines, it is examined by addres decoders of all devices on the bus.



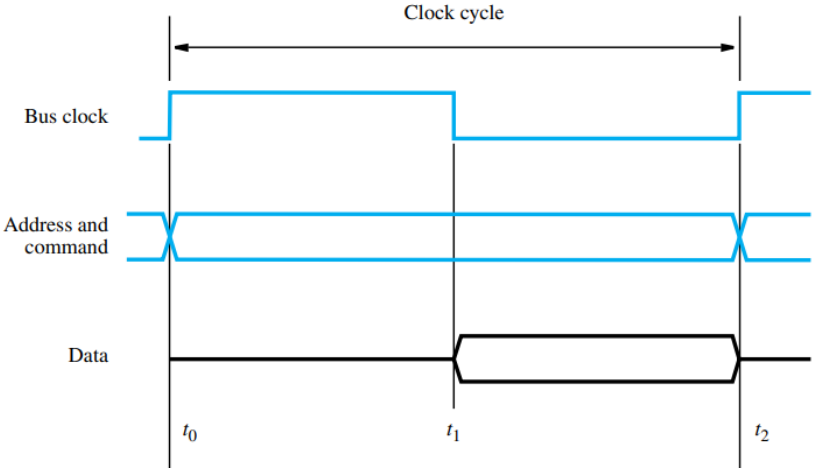
I/O interface for an input device.

- The device that recognizes this address responds to the commands issued on the control lines.
- The processor uses the control lines to request either a Read or a Write operation
- The requested data are transferred over the data lines.
- When I/O devices and the memory share the same address space, the arrangement is called memory-mapped I/O
- **interface circuit:** the device's address decoder, data and status registers, and the control circuitry required to coordinate I/O transfers.

BUS OPERATION

- **bus protocol:** set of rules that govern how the bus is used by verious devices. It defines when a device may place information on the bus, when it may load data on the bus, etc. all done by control signals (such as R/$\overline{W}$)
- The bus control lines also carry **timing information**. They specify the times at which the processor and the I/O devices may place data on or receive data from the data lines. There **syncronous** and **asyncrhonous**.
- **master:** devide that initates data transfer by issuing Read or Write commands on the bus. (often the CPU)
- **slave:** the deviced addressed by the master.

SYNCHRONOUS BUS



- **clock cycle:** clock signal's two phases: the **high level** followed and the **low level** that follows.
- **clock pulse:** first half of the cycle (the high part).
- **diamond:** means change in value.
- **halfway line:** unreliable/ignored data.

Single bit, switching between 0 and 1

Multiple bits, first all 0s, than mix of 1s and 0s

Multiple bits, mix of 1s and 0s, changing over time

Read operation:

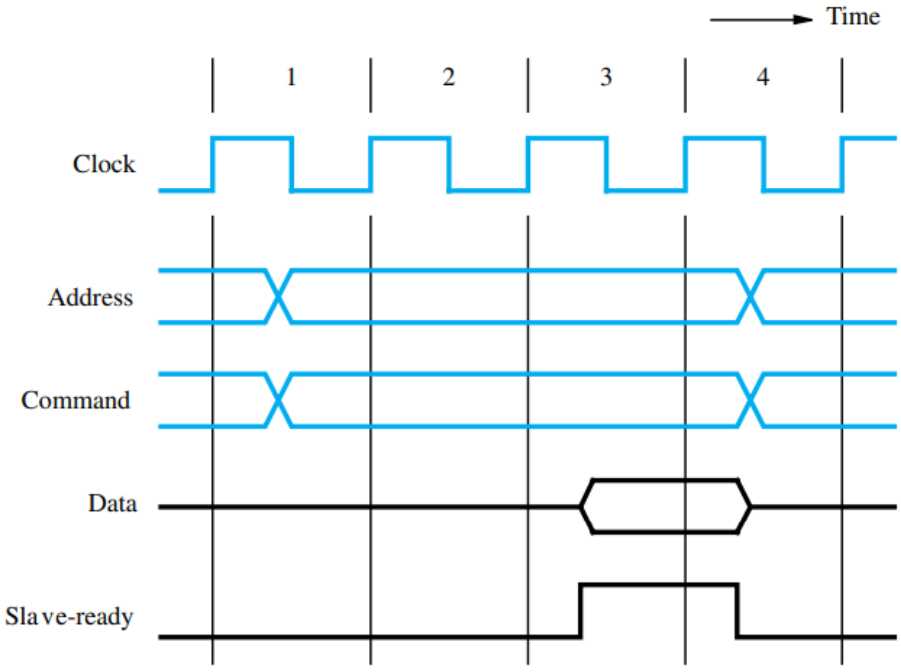
1. **Master** places the **slave address** on the addres lines and sends a **command** on the control lines.
2. The **clock bus period from t0 to t1** > the **maximum propagation delay over the bus**. (long enough to allow step 3).
3. **All devices decode the address** and control signals, and **only** the **slave** places at t1 the requested input **data** on the data lines.
4. At the end of the clock cycle, at time t2, the **master loads the data** on the data lines into one of its registers. The **clock bus periodf from t1 to t2 > (t0 to t1) + setup time of the master's register**.

Write operation:

1. **Master** places the **slave address** on the addres lines a **command** on the control lines and the **data** on the data line.
2. The **clock bus period from t0 to t1** > the **maximum propagation delay over the bus**. (long enough to allow step 3).
3. **All devices decode the address** and control signals, and **only** the **slave,** at t1 loads the output **data** on into its data register
4. The **clock bus periodf from t1 to t2 > (t0 to t1) + setup time of the slave's register**.

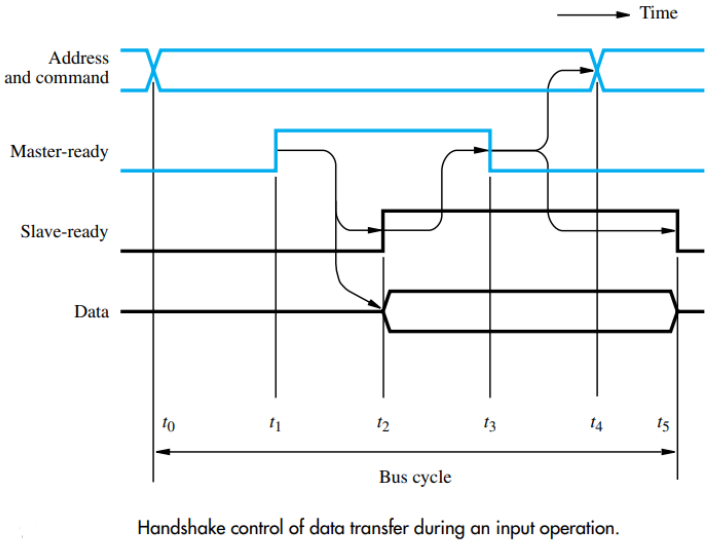
Because of propagation delays on bus wires and in the circuits of the devices, while the clock changes are assumed to be seen at the same time by all devices connected to the bus:

- **a given signal transition is seen by different devices at different times**.
- This **forces all devices to operate at the speed of the slowest device**.
- **solution:** bus incorporates **control signals** (Slave-ready) that represent the **response** from the device. These signals inform the master that the slave has recognized its address and that it is ready to participate in a data transfer operation. The number of **clock cycles will vary** from one device to another.
- The **master**, which has been waiting for this signal, **loads** the data into its register **at the end of the clock cycle**
- Save removes its data signal from the bus and returns its slave-ready signal to the low level by end of cycle T
- the master may send new address and command signals to start a new transfer in clock cycle T+1
- If the addressed device does not respond at all, the master waits for some predefined maximum number of clock cycles, then aborts the operation (i.e. wrong address or device malfunction)



An input transfer using multiple clock cycles.

ASYNCHRONOUS BUS



Handshake control of data transfer during an input operation.

Aka **handshake** protocol: exchange of command and response signals between the master and the salve (so no need for Slave-ready & clock (negative) edge). A control line Master-ready is asserted by the master to indicate that it is ready to start a data transfer.

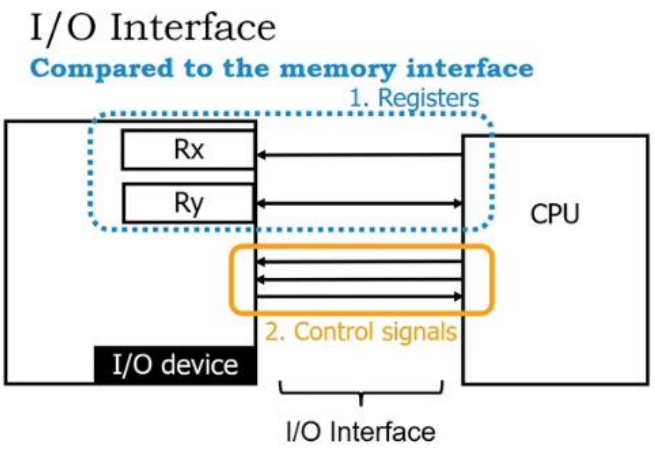
1. The master places the address and command information on the bus.
2. Then it indicates to all devices that it has done so by activating the Master-ready line.
3. This causes all devices to decode the address
4. The selected slave performs the required operation and informs the processor that it has done so by activating the Slave-ready line.
5. The master waits for Slave-ready to become asserted before it removes its signals from the bus. In the case of a Read operation, it also loads the data into one of its registers.

- **fully interlocked/full handshake:** a change in one signal is always in resonse to a chinge in the other. Highest degree of flexibility and reliabilit
- **advantage:** the handshake protocol eliminates the need for distribution of a single clock signal whose edges should be seen by all devices at about the same time (simplifies the design), plus delays are flexible, whereas in synchronous you will be bottlenecked by the slowest device.
- **disadvantage:** it is only advantageous when there are slow devices. If all devices are within the same range it is better to use a synchronous clock because you will only need to accomadate a one round trip delay instead of two.

ARBITRATION

- **bus driver:** A logic gate that places data on the bus.
- **arbitrer circuit (round robin scheme):** circuit that decides who uses a specific resource request by multiple entities at once. The arbitrer associates priorities with individual **requests**. It will **grant** it to higher priority first. Once the driver is done, it deactivates its Bus-grant.

I/O LECTURE

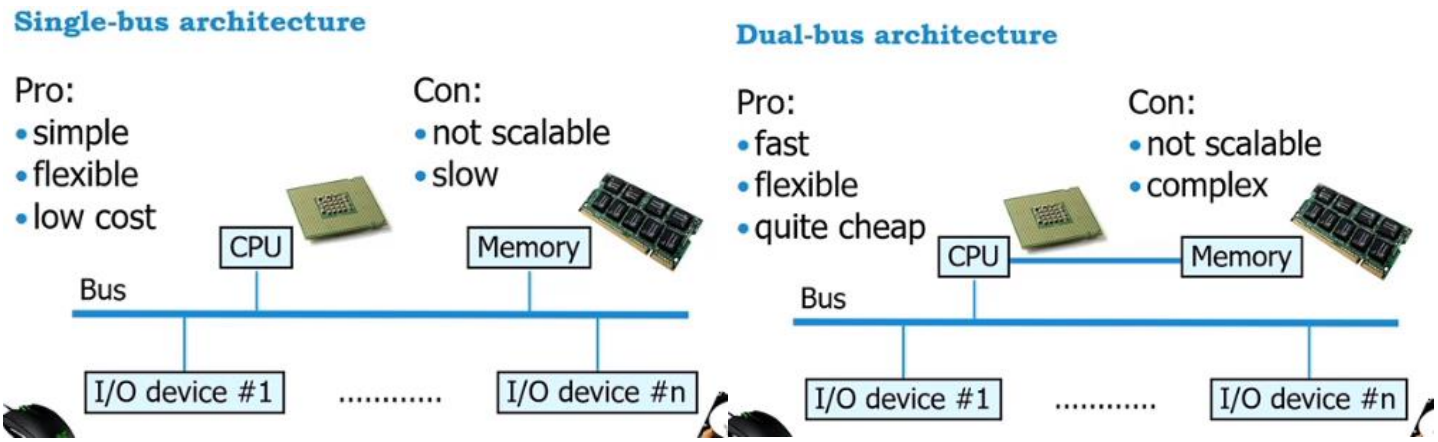


Interface: Information exchange protocol between elements, ISA is an interface between hardware and software.

Interfaces are portable, so that the same protocol works with different elements.

The difference between the Memory – CPU interface and the Device CPU interface is that the Devices have registers inside them, so that the CPU can access them just as if they were normal registers, thouse would be mapped registers.

1. Data registers
   - to store incoming and outgoing data
2. Status and control registers
   - to certify status of device
   - to control transfer



- Black thick lines are buses

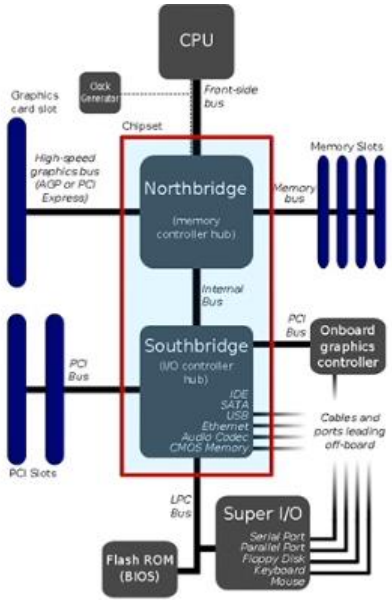
I/O organization

A typical motherboard

Northbridge
- high performance

Southbridge
- flexibility

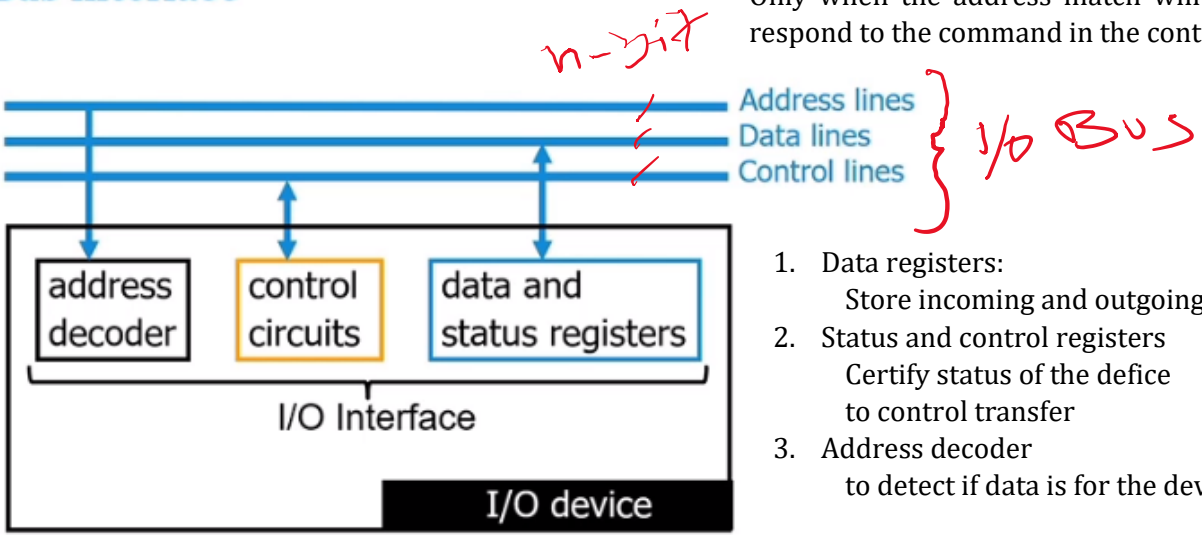
Other chipsets:
- Sandy Bridge (2011—)
- Intel Skylake (2015—)



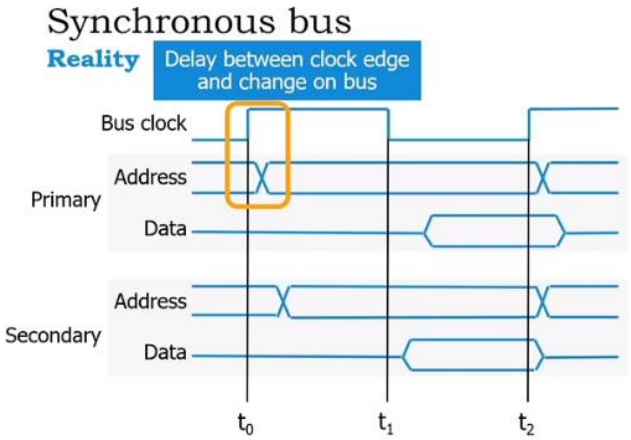
- North bridge focuses on performance (it has quick access to the CPU) Main memory and GPU are there.
- South bridge is for all other devices (connectivity, USB, ethernet, keyboard, periferals) and provides felxibility
- North and South combined = computer chipset. Component on the mother board that connects the CPU to all the other devices that we could potentially connect to.
- You can't plug any CPU in any chipset as these work with a very unique chemistry (to gain efficency) there is not a fixed interface between a CPU and a general chipset.
- However, the chipset itself is compatible with a lot of devices

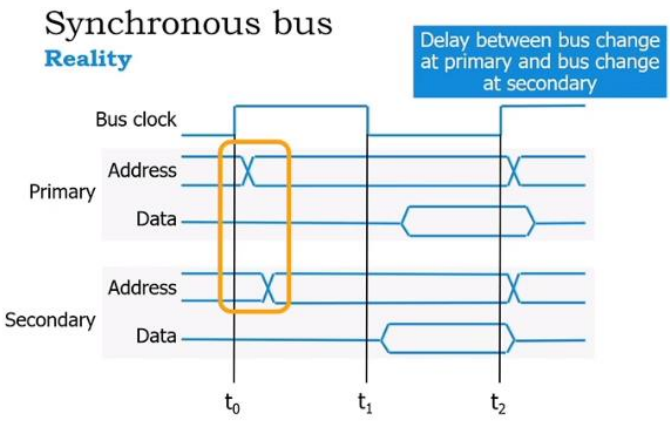
Bus interface

Only when the address match will the slave device respond to the command in the control lines.



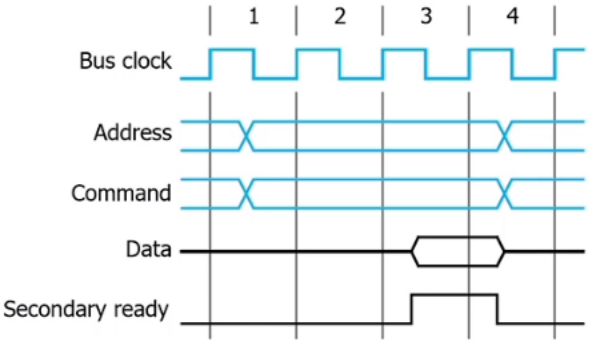
1. Data registers:
   Store incoming and outgoing data
2. Status and control registers
   Certify status of the defice
   to control transfer
3. Address decoder
   to detect if data is for the device





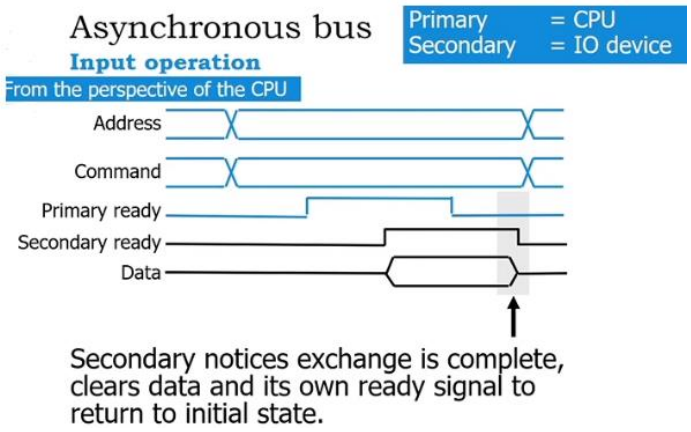
- set up time (lots of gates to change values)
- propogation delay
- drawback: slowest device sets the delay adjustment clock speed for all devices

Multi-cycle bus – handle device variability



faster devices need fewer cycles to respond

signal completed to CPU



Explicit handshaking (vs synchronous clock)
Timing must account for signal propagation skew, caused by detours (longer paths (wire length and gates))

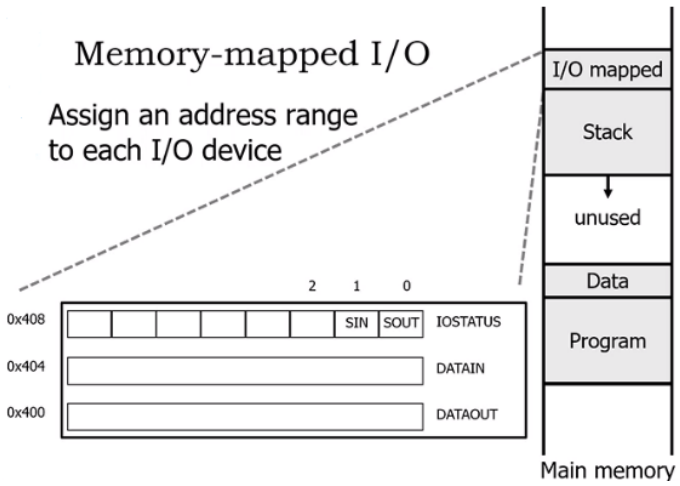
How does the CPU talk to IO devices?

Port-mapped I/O

1. Use separate memory space for I/O devices.
2. Use special instructions to interact with this separate memory space.

Memory-mapped I/O

1. Map I/O devices to region in main memory.
2. Use existing instructions to move data to/from device.



Not all of the 2^64 bits of the main memory addresses will be used so some are allocated to device registers.

So you will map specific addresses, in hex generally
Therefore the same move command can be used interchangeably with registers both inside and outside the CPU

- MOV DATAIN, R1
  # read from the keyboard
- MOV $42, DATAOUT
  # write to the screen

CPU executes the I/O program

- Unconditional I/O
  - no synchronization with I/O device
- Passive signaling (Polling)
  - synchronization between CPU and I/O device by programmed interrogation by CPU
- Active signaling (Interrupts)
  - synchronization between CPU and I/O device by active interrupt of device

Programming I/O routines:

Use status registers and **busy waiting**

```
READWAIT:   Testbit  #1, IOSTATUS
            Branch=0  READWAIT
            Move DATAIN, R1
```

```
WRITEWAIT:  Testbit  #0, IOSTATUS
            Branch=0  WRITEWAIT
            Move R1, DATAOUT
```

Actively wait in a loop until the bit is 1.
Keeps the CPU busy with this loop.

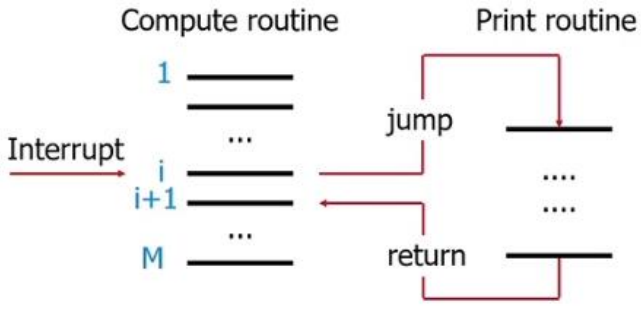
Unconditional I/O. The device is constantly sending data but the CPU reads it whenever he wants. You get racing the beam issue: The pixels are displayed when the CPU wants, so half of the image has old pixels the other half new pixels

Passive signaling (polling) : Similar as buys waiting, but instead of every cycle it is every x seconds
Active signaling: removes control from the cpu.

The CPU may not be interrupted when it is executing an instruction, these are atomic. The interrupt will sneak in the PC after the current instruction is completed.

Interrupts

Programmer's perspective



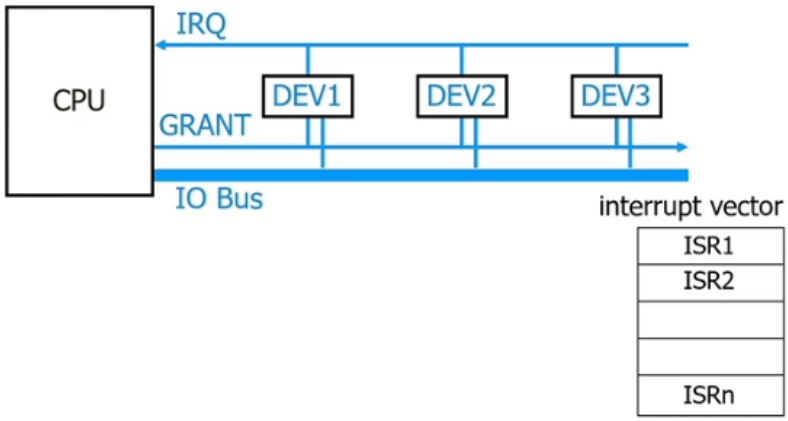
Control passes to other code on an external event
- ISR – Interrupt Service Routine (aka interrupt handler)
- at random (!) moment in time

Interrupts

Execution flow

1. I/O device alerts CPU by hardware signal
   - Interrupt ReQuest line (IRQ, part of the I/O bus)
2. CPU stops program execution
3. Interrupts are **disabled**
4. Device is informed of acceptance and clears IRQ
5. ISR is invoked to handle device's request
6. Interrupts are **enabled**
7. Execution of program resumes

Vectored interrupts



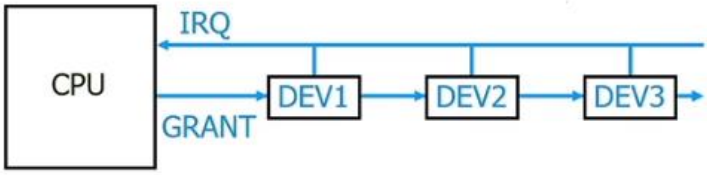
Shared interrupt line hardware to the CPU, which the CPU cant ignore.

1. A Device sends interrupt signal via the IRQ
2. CPU sends Grant signal back to (all) devices via GRANT (different when prioritizing)
3. Device sends ID on data bus
4. CPU calls ISR from interrupt vector [ID]

The interrupt vector information is usually in the device drivers.

Daisy chaining = hooking up multiple devices in a sequence.

Daisy chain



When multiple devices raise a grant signal the priority will go from the closest device to the CPU to the furthest. (Because you are the first to receive the grant signal).

If the high priority device didnt raise the interrupt it will pass the grant on to the next devices.

- Rule out simultaneous interrupts by prioritizing devices
- Fixed order (from left to right)
  - when inactive, pass signal on

Recap

1. Users generate increasingly more data. We need **high-performance** and **flexible** I/O to meet demand
2. I/O devices use an **interface** based on signals and registers
3. I/O devices are connected to the CPU using one or multiple **(a)synchronous buses**
4. If the CPU controls the I/O devices:
   1. We can use **memory-mapped** or **port-mapped** I/O
   2. We can use **unconditional I/O**, **busy-waiting**, **polling**, or **interrupts**
5. **Vectored interrupts**/**daisy-chaining** support multiple devices/interrupts, different characteristics

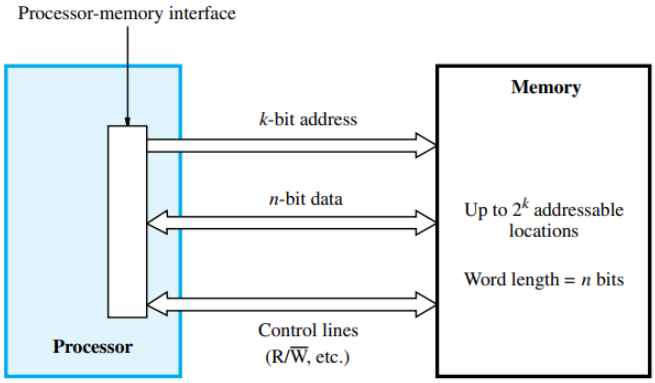
CHAPTER 8 MEMORY (SOLVED PROBLEMS PAGE 324)

The memory of a computer comprises a hirearchy, including a cache, the main memory, and secondary storage.

Direct memory access is a mechanism to **transfer data between an I/O device**, such as a disk, **and the main memory** with **minimal involvement from the processor**.
Caches decrease memory access times.

BASIC CONCEPTS

- The maximum size of the memory that can be used in any computer is determined by the addressing scheme.
- 16-bit addresses can have 2^16 memory locations.
- Memory is usally designed to store and retrieve data in word-length quantities. From now own, assume 32-bit addresses for byte addressable memories. The high order 30 bits determine which word is specified. the low order 2 bits of the address specify which byte location is involved.
    - A word is 2 bytes.
    - long is 4 bytes
    - quad is 8 bytes
- The connection between the CPU and memory consist of:
    - addres
    - data
    - control lines
- The processor uses the addres lines to specify the memory location involved in a data transfer operation
- The processor uses the data liens to transfer the data at such specific address
- The control lines carry the command indicating a Read or Write operation and whether a byte or a word is to be transferred.



Connection of the memory to the processor.

- Control lines also provide the timing information by asserting MFC
- **memory access time:** speed of memory unit that elapses the time between initation of an operation to transfer a word of data and the completion of that operation.
- **memory cycle time:** minimum **time** delay required **between** the initation of **two successive memory operations** (i.e. time between 2 successive reads). Cycle time is usually shorter than access time.
- **random-access memory (RAM):** if **the access time to any location is the same**, independent of the location's address. Which is different to other type of memories such as disc, where certain data is located at places that take longer for the disc to read. The cycle times range from 100ns to less than 10ns

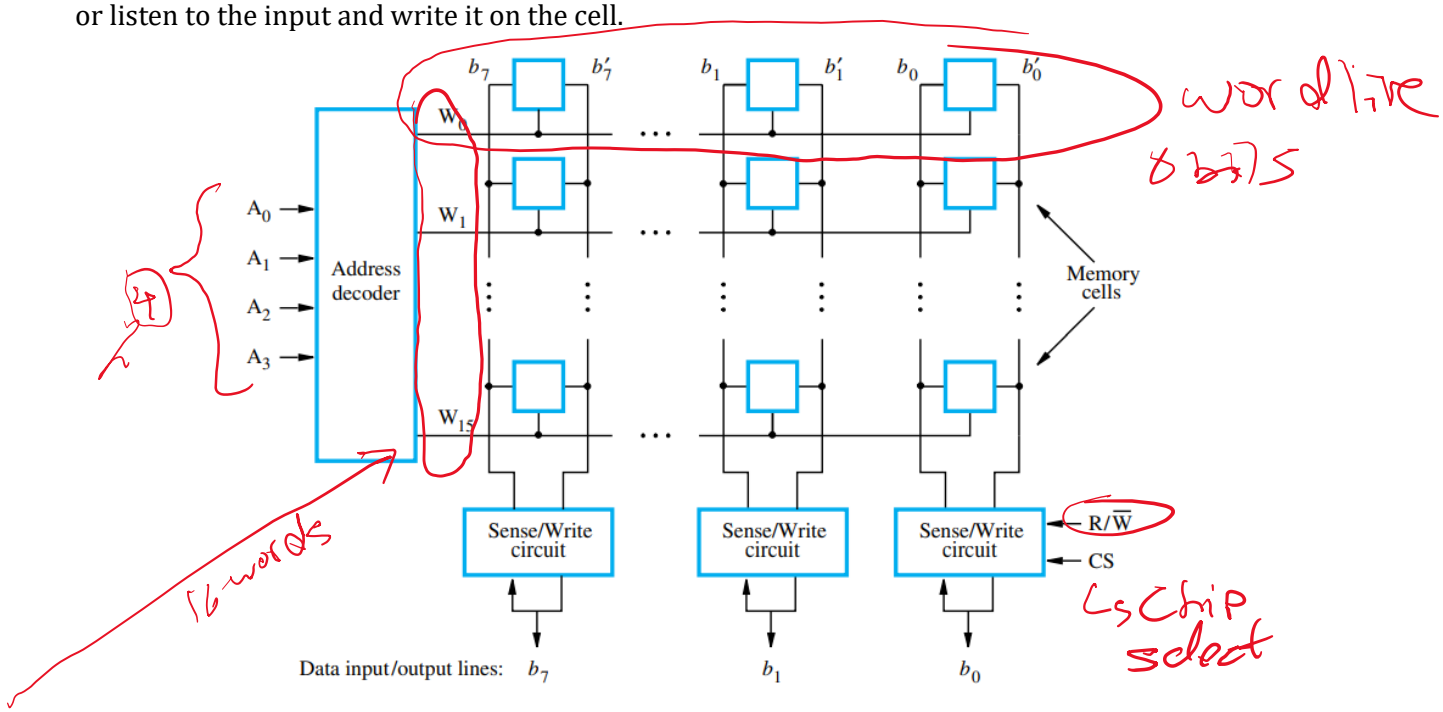
CACHE AND VIRTUAL MEMORY

- **Memory access time is the bottleneck** in the CPU as decoding and processing the instruction take less time than fetching the instruction from memory.
- **cache memory:** small fast memory inserted **between** the larger (slower) **main** memory and the **processor**. It holds the currently **active portions of a program and its data**.
- **virtual memory:** only the active portions of a program are stored in the main memory, and the remainder is tored on the much larger secondary device. Sections of the program are transferred back and forth. Therefore the application sees a memory that is much larger than the computer's actual main memory.
- **block transfers:** Data is transfered in blocks involving tens to thousands of words
Semiconductor RAM Memories

INTERNAL ORGANIZATION OF MEMORY CHIPS

- Memory **cells** are commonly organized in the form of an **array**, where each **cell** stores **1 bit** of data
- Each row constitues a **memory word**
- **word line:** cells of a row are connected to a common line. Driven by the **address decoder** on the chip

- columns are connected to a Sense/Write circuit by **2 bit lines** which are connected to the data input/output lines of the chip. Depending on the Read/Write signals the Sense/Write will either output the cells contents or listen to the input and write it on the cell.



16 words of 8 bits each (it is still up to the architect to define how many bits a word contains). This is refered to as a 16x8 organization. The configuration above stores 128 bits (16*8) and reqiores 14 external connections for address, data and control lines. It also needs 2 lines for power supply and ground conections.

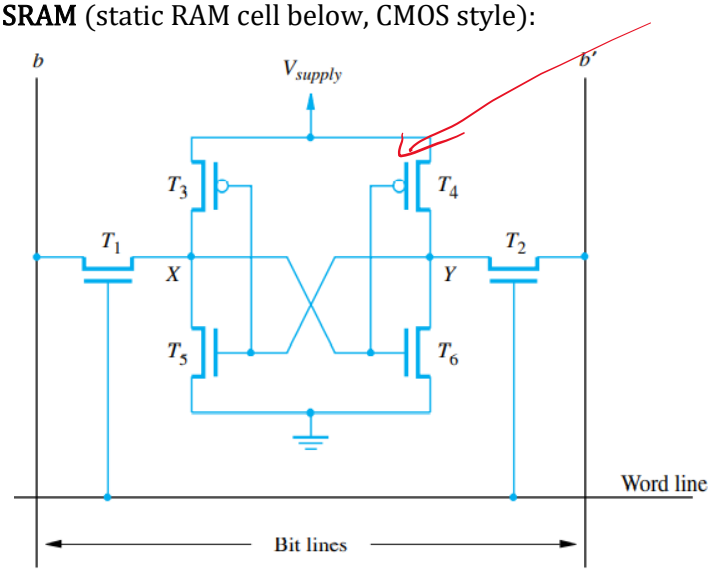
1024 cells, organized as a 128x8 memory. Requires a total of 19 external connections.
1kx1 setting would be a 10 bit address, with only one data line, resulting in 15 external connections

a 1G-bit chip may have a 256M × 4 organization, in which case a 28-bit address is needed and 4 bits are transferred to or from the chip

STATIC MEMORIES

static memories: memories that consist of circuits capable of retaining their state as long as power is applied.

SRAM (static RAM cell below, CMOS style):

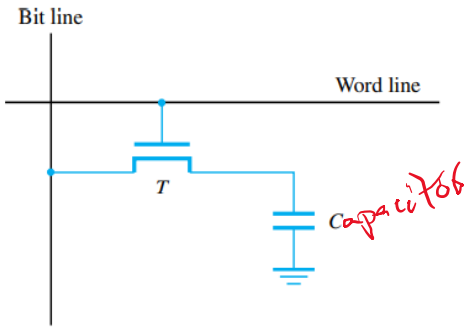


- Two inverters are cross-connected to form a latch
- The latch is connected to two bit lines by transistors T1 and T2
- Transistors act as switches, which are controlled by the word line. They are NMOS so the represent the exact same value that the word line has. if low they are open (so latch retians its current value) if high they are closed.
- to read the state of the SRAM cell, the word line is activated to close switches T1 and T2
- Cell = 1 if b1 high, b' low.
- Cell = 0 if b1 low, b' high
- To write, the Sense/Write circuit drives bit lines b and b' , instead of sensing their state. It sets b and b' accordingly and actiates the word line to save it (once disabled, the latch will keep the current value.

- Continues power is needed for the cell to retain its state.

- When power is restored after an interruption, the latch settles into a stable but not necessarily the same as the last state, this makes SRAM **volatile**, because their contents are lost after power is gone.
- **advantage:** low power consumption (current flows in the cell only when the cell is being accesssed. There is no connection between supply and ground but the state is kept. Another advantage is that they can be accessed very quickly (ns)

DYNAMIC RAMS

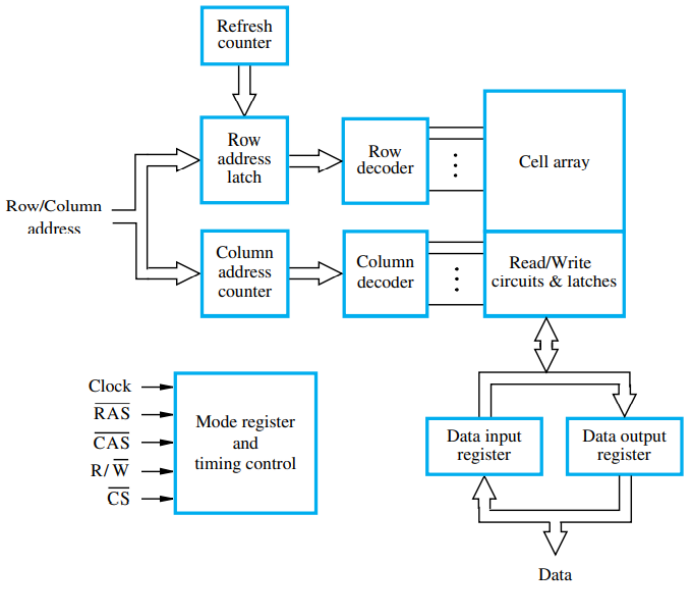


A single-transistor dynamic memory cell.

DRAMS (dynamic RAMs): Less expensive and higher density RAMs implemented with simpler cells that can't retain their state for a long period unless they are accessed frequently.

- Information is stored in a dynamic memory cell in the form of a charge on a capacitor, but this charge can be maintained **for only tens of milliseconds**.
- its contents must be periodically refreshed by restoring the capacitor charge to its full value (this occurs when the contents of the cell are read or written into it).
- To store information in this cell, transistor T is turned on and an appropriate voltage is applied to the bit line
- After the transistor is turned off, the charge remains stored in the capacitor, but not for long as the capacitor begins to discharge after is totally turned off.
- A sense amplifier connected to the bit line detects whether the charge stored in the capacitor is above or below the threshold value
- If the charge is above the threshold, the sense amplifier drives the bit line to the full voltage representing the logic value 1. As a result, the capacitor is recharged to the full charge corresponding to the logic value 1.
- If the sense detects the capacitor below the threshold, it pulls the bit line to ground level to discharge the capacitor fully.
- Since the word line is common to all cells in a row, all cells in a selected row are read and refreshed at the same time after reading the contents of a single cell of that row.
- **Row Address Strobe (RAS signal)**: input control line that causes a read operation to be initiated, in which all cells in the selected row are read and refreshed.
- **fast page mode feature:** a block of data (often called page) transfered at a much faster rate by applying a consecutive sequence of column addresses (**CAS signals = Column Address Strobe**).

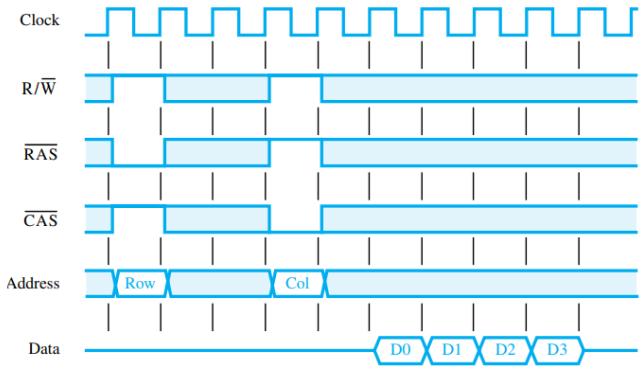
SYNCHRONOUS DRAMS



Synchronous DRAM.

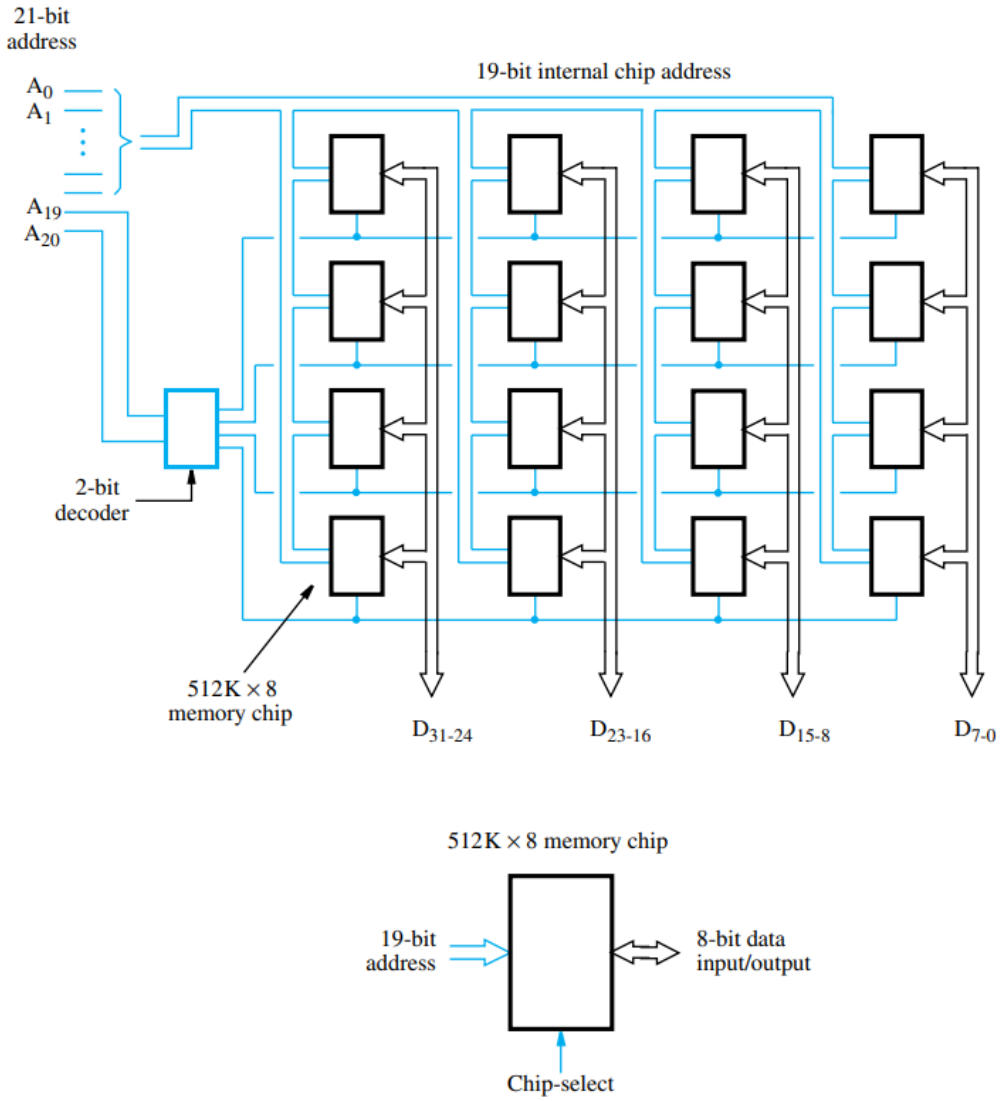
- DRAMS syncrhonized by a clock signal.
- Clock sends refreshing signal to selected rows, which makes the dynamic nature of these memory chips is almost invisible to the user.
- SDRAMs have several different modes of operation, which can be selected by writing control information into a mode register
- burst operations of different lengths can be specified

New data are placed on the data lines at the rising clock



- The column address is latched under control of the CAS signal
- Synchronous DRAMs can deliver data at a very high rate, because all the control signals needed are generated inside the chip.
- Today's SDRAMs operate with clock speeds that can exceed 1 GHz
- **memory latency:** time it takes to transfer the first word of a block
- **memory bandwidth:** performance measure: number of bits or bytes that can be transferred in one second
- **Double_data_rate SDRAM:** To make the best use of the available clock speed, data are transferred externally on both the rising and falling edges of the clock
- **Rambus Memory:** The key feature of Rambus technology is the use of a **differential-signaling technique** to transfer data to and from the memory chips.

STRUCTURE OF LARGER MEMORIES



Organization of a 2M × 32 memory module using 512K × 8 static memory chips.

The R/W inputs of all chips are tied together to provide a common Read/Write control lin (not shown in the figure)

DYNAMIC MEMORY SYSTEMS

- A large memory leads to better performance, because more of the programs and data used in processing can be held in the memory, thus reducing the frequency of access to secondary storage
- Because of their high bit density and low cost, **synchronous dynamic RAMs**, are widely used in the memory units of computers

- They are slower than static RAMs, but they use less power and have considerably lower cost per bit
- **Memory modules** are commonly called **SIMMs** (Single In-line Memory Modules) or **DIMMs** (Dual In-line Memory Modules)

MEMORY CONTROLLER

- The address applied to dynamic RAM chips is divided into two parts:
  - high-order address bits: select a row in the cell array
    (provided first and latched into the memory chip under control of the RAS signal)
  - lower-end bits: select a column
    (provided on the same address pins and latched under control of the CAS signal)
- Since a typical processor issues all bits of an address at the same time, a multiplexer is required (**memory controller circuit**)

REFRESH OVERHEAD

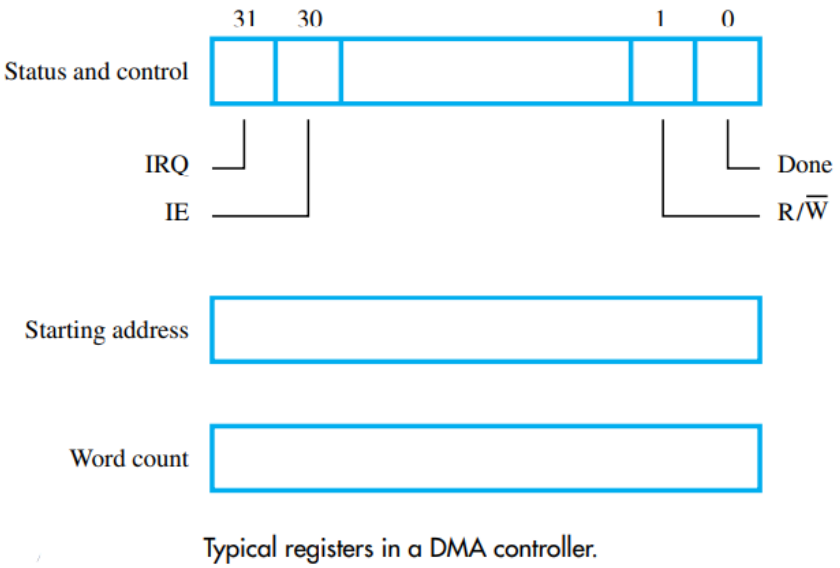
- A dynamic RAM cannot respond to read or write requests while an internal refresh operation is taking place
- Such requests are delayed until the refresh cycle is completed
- the time lost to accommodate refresh operations is very small

consider an SDRAM in which each row needs to be refreshed once every 64 ms. Suppose that the minimum time between two row accesses is 50 ns and that refresh operations are arranged such that all rows of the chip are refreshed in 8K (8192) refresh cycles. Thus, it takes $8192 \times 0.050 = 0.41$ ms to refresh all rows. The refresh overhead is $0.41/64 = 0.0064$, which is less than 1 percent of the total time available for accessing the memory.

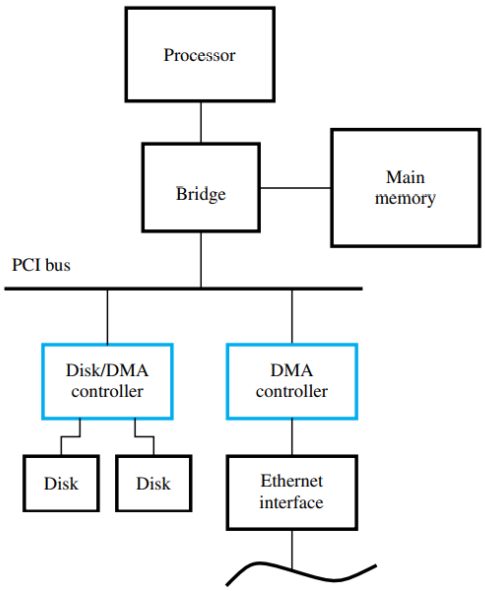
COMPARING RAMS

- Static RAMs (**SRAM**) are used where a small but very fast memory is needed (**cache**)
- Dynamic RAMs are cheaper and have high bit density
- Synchronous Dynamic Rams (**SDRAM**) are the better version of DRAM and used for the **main memory**

DIRECT MEMORY ACCESS



Typical registers in a DMA controller.



Use of DMA controllers in a computer system.

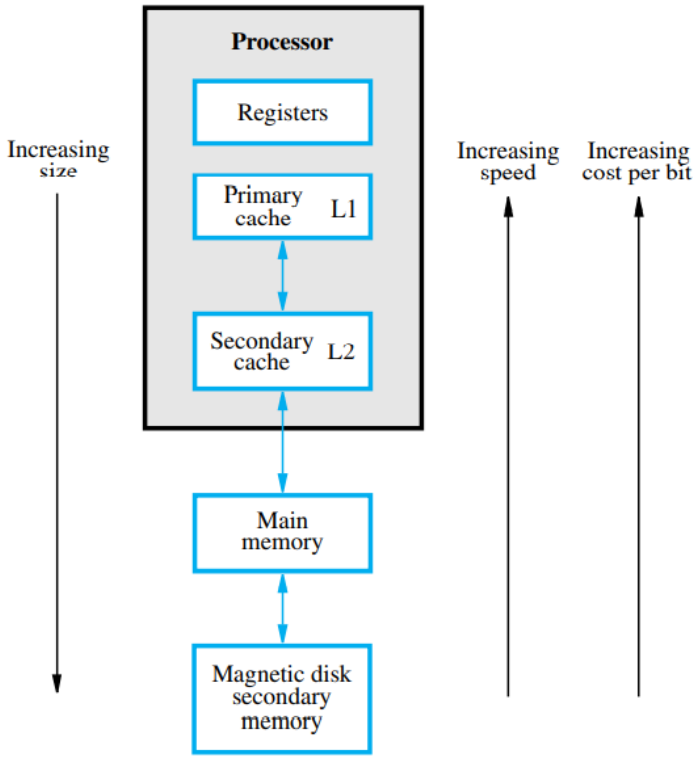
Data are transferred from an I/O device to the memory by first reading them from the I/O device using an instruction such as: LOAD R2, DATAIN //DST, SRC

- Considerable overhead is incurred, because several program instructions must be executed involving many memory accesses for each data word transfered.

- **direct memory access (DMA):** An alternative approach is used to transfer blocks of data directly between the main memory and I/O devices, such as disks.
- The unit that controls DMA transfers is referred to as a DMA controller which performs the functions that would normally be carried out by the processor when accessing the main memory
- Although a DMA controller transfers data without intervention by the processor, its operation must be under the control of a program executed by the processor, usually an **operating system routine**.
- To initiate the transfer of a block of words, the processor sends to the DMA controller the starting address, the number of words in the block, and the direction of the transfer.
- The DMAcontroller then proceeds to perform the requested operation. When the entire block has been transferred, it informs the processor by raising an interrupt.
- Two registers are used for storing the starting address and the word count. The third register contains status and control flags.
- Done flag 1 iWhen controller completes transferring a block of data and is ready to receive another command
- Bit 30 is the Interrupt-enable flag, IE. When this flag is set to 1, it causes the controller to raise an interrupt after it has completed transferring a block of data.
- The controller sets the IRQ bit to 1 when it has requested an interrupt.

MEMORY HIERARCHY

- An ideal memory would be fast, large, and inexpensive
- a very fast memory can be implemented using static RAM chip
- these chips are not suitable for implementing large memories, because their basic cells are larger and consume more power than dynamic RAM cells.
- Although dynamic memory units with gigabyte capacities can be implemented at a reasonable cost, the affordable size is still small compared to the demands of large programs with voluminous data.
- A solution is provided by using secondary storage, mainly magnetic disks, to provide the required memory space. Disks are available at a reasonable cost, and they are used extensively in computer systems. However, they are much slower than semiconductor memory units.
- affordable, (smaller) main memory can be built with dynamic RAM technology.
- static RAM technology to be used in smaller units where speed is of the essence, such as in cache memories. This memory, called a **processor cache** holds copies of the instructions and data from the main memory,
- The fastest access is to data held in processor registers.



Memory hierarchy.

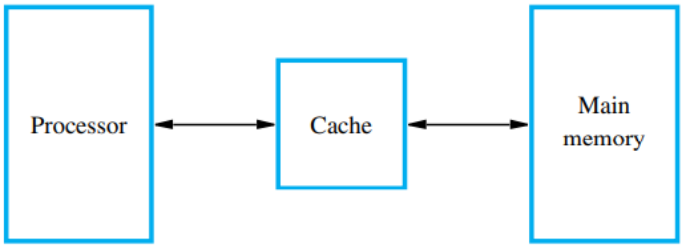
A primary (L1) cache is always located on the processor chip. This cache is small and its access time is comparable to that of processor registers

A larger, and hence somewhat slower, secondary (L2 sometimes even L3) cache is placed between the primary cache and the rest of the memory. Often it is also housed on the processor chip.

CACHE MEMORIES

- The cache is a small very fast memory between the processor and the main memory.
- **locality of referene:** approach to make the main memory appear to the processor to be much faster than it is.
- most of program execution time is spent in routines in which many instructions are executed repeatedly.
- a recently executed instruction is likely to be executed again very soon

- instructions close to a recently executed instruction are also likely to be executed soon



- **property of locality of reference**: whenever an information item, instruction or data, is first needed, this item should be brought into the cache, because it is likely to be needed again soon.
- Spatial locality suggests that instead of fetching just one item from the main memory to the cache, it is useful to fetch several items that are located at adjacent addresses as wel
- **cache block/cache line:** set of contiguous address locations of some size
- **mapping function:** specifies the correspondence between the main memory blocks and those in the cache
- **replacement algorithm:** cache control hardware that decides which block should be removed to create space for the new block that contains the referenced word
- **cache hits:** the processor without knowing whether the issued address is cached or not goes through the **cache control circuitry** and if it does a **read** or **write hit** occurs.
- **read hit:** the main memory is not involved
- **write hit:** option 1, **write-through protocol**, both cache and main memory are updated. Option 2,
- **write-back/ copy-back:** only the cache location is updated and marks the block containing it with an associated fal bit (**dirty/modified bit**) and right before the cache word is going to be removed for a new block the main memory location of the word is updated.
- The write-through protocol is simpler than the write-back protocol, but it results in unnecessary Write operations in the main memory when a given cache word is updated several times during its cache residency.
- The write-back protocol also involves unnecessary Write operations, because all words of the block are eventually written back, even if only a single word has been changed while the block was in the cache. Still write-back is used most often as it takes advantage of the data block transfer efficency.
- **cache misses:** when a word is not found in the cache. Which will copy the main memory words to the cache.
- **load-through/early restart:** it first sends the word to the processor and then to the cache to reduce processor's waiting time at the spend of more complex circuitry.
- **Write miss** with **write-through** protocol: the information is written directly into the main memory.
- **Write miss** with **write-back protocol**: the block containing the addressed word is first brought into the cache, and then the desired word in the cache is overwritten with the new information.
- many processors use separate caches for instructions and data, making it possible for the two operations to proceed in parallel.
- **direct mapping:** block j of the main memory maps onto block j modulo 128 of the cache. Contention is resolved by allowing the new block to overwrite the currently resident block. With direct mapping, the replacement algorithm is trivial. Placement of a block in the cache is determined by its memory address. The direct-mapping technique is easy to implement, but it is not very flexible.

Tag	Block	Word
5	7	4

Block position	Contents of data cache after pass:								
	$j = 1$	$j = 3$	$j = 5$	$j = 7$	$j = 9$	$i = 6$	$i = 4$	$i = 2$	$i = 0$
0	A(0,0)	A(0,2)	A(0,4)	A(0,6)	A(0,8)	A(0,6)	A(0,4)	A(0,2)	A(0,0)
1									
2									
3									
4	A(0,1)	A(0,3)	A(0,5)	A(0,7)	A(0,9)	A(0,7)	A(0,5)	A(0,3)	A(0,1)
5									
6									
7									

Contents of a direct-mapped data cache.

- **associative mapping:** the most flexible mapping meppod, a main memory block can be placed into any cache block position. It has a more fficient use of the space in the cache. When a new block is brought into the cache, it replaces (ejects) an existing block only if the cache is full. **Associative search** searches the tags in parallel.

Tag	Word
12	4

Block position	Contents of data cache after pass: $j = 7$	$j = 8$	$j = 9$	$i = 1$	$i = 0$
0	A(0,0)	A(0,8)	A(0,8)	A(0,8)	A(0,0)
1	A(0,1)	A(0,1)	A(0,9)	A(0,1)	A(0,1)
2	A(0,2)	A(0,2)	A(0,2)	A(0,2)	A(0,2)
3	A(0,3)	A(0,3)	A(0,3)	A(0,3)	A(0,3)
4	A(0,4)	A(0,4)	A(0,4)	A(0,4)	A(0,4)
5	A(0,5)	A(0,5)	A(0,5)	A(0,5)	A(0,5)
6	A(0,6)	A(0,6)	A(0,6)	A(0,6)	A(0,6)
7	A(0,7)	A(0,7)	A(0,7)	A(0,7)	A(0,7)

Contents of an associative-mapped data cache.

	Contents of data cache after pass: $j = 3$	$j = 7$	$j = 9$	$i = 4$	$i = 2$	$i = 0$
Set 0	A(0,0)	A(0,4)	A(0,8)	A(0,4)	A(0,4)	A(0,0)
	A(0,1)	A(0,5)	A(0,9)	A(0,5)	A(0,5)	A(0,1)
	A(0,2)	A(0,6)	A(0,6)	A(0,6)	A(0,2)	A(0,2)
	A(0,3)	A(0,7)	A(0,7)	A(0,7)	A(0,3)	A(0,3)
Set 1						

Contents of a set-associative-mapped data cache.

- **set associative mapping:** combiation of direct and associative mapping. Main memory blocks may reside in any set. It eases the block rplacement problem of direct mapping. Associative search is also reduced.

Tag	Set	Word
6	6	4

- **stale data:** when the power is turned on, the cache contains no valid data (stale) (so all valid bits are reset to 0). A control bit (**valid bit**) must be provided to tell whether the block data is valid or not. The processor fetches data from a cache block only if its valid bit is equal to 1. So as program execution proceeds, the valid bit of a given cache block is set to 1 when a memory block is loaded into that location.
- **flush the cache:** forces all dirty blocks to be written back to the memory before performing the transfer.
- **cache-coherence problem:** the need to ensure that two different entities (the processor and the DMA subsystems in this case) use identical copies of the data.
- **replacement algorithms:**
  - In a direct-mapped cache, the position of each block is predetermined by its address; hence, the replacement strategy is trivial. For the rest:
  - **least recently used (LRU) replacent algorithm:** overwrite the block that has gone the longest time without being referenced (a 2-bit counter can be used for each block).
    - Performance of the LRU algorithm can be improved by introducing a small amount of randomness in deciding which block to replace.
  - "**oldest**" block from a full set when a new block must be **brought in**. This algorithm does not take into account the recent pattern of access to blocks in the cache, it is generally not as effective as the LRU algorithm
  - the simplest algorithm is to **randomly choose the block to be overwritten**. Interestingly enough, this simple algorithm has been found to be **quite effective** in practice.

PERFORMANCE CONSIDERATIONS

- **Hit rate and miss penalty**: Consider a system with only one level of cache. In this case, the miss penalty consists almost entirely of the time to access a block of data in the main memory. Let h be the hit rate, M the miss penalty, and C the time to access information in the cache. Thus, the average access time experienced by the processor is: $t_{avg} = hC + (1 - h)M$
- Hit rate and miss **with 2 cache**: $t_{avg} = h_1C_1 + (1 - h_1)(h_2C_2 + (1 - h_2)M)$
- **number of misses in the L2 cache**: $(1 - \text{h1})(1 - \text{h2})$

h_1 is the hit rate in the L1 caches.

h_2 is the hit rate in the L2 cache.

C_1 is the time to access information in the L1 caches.

C_2 is the miss penalty to transfer information from the L2 cache to an L1 cache.

M is the miss penalty to transfer information from the main memory to the L2 cache.

= cache + memory time

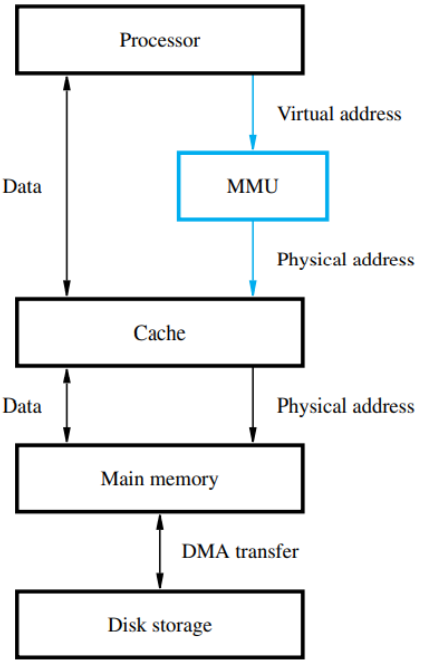
you don't know a priori if it is in cache, so you always check there first

- **write buffer:** writing tasks can be delayed and performed in bulks (buffers) because the processor doesn't usually need to access the written data immeditely again, for reading requests is the opposite.
- **prefetching:** To avoid stalling the processor, it is possible to prefetch the data into the cache before they are needed. A special prefetch instruction may be provided in the instruction set of the processor. They can be inserted into a program either by the programmer or by the compiler.
- **lockup-free cache:** prefetching can lock the entire cache room. Lockup free cache allows the processor to access the cache and have more than one outstanding miss.

VIRTUAL MEMORY

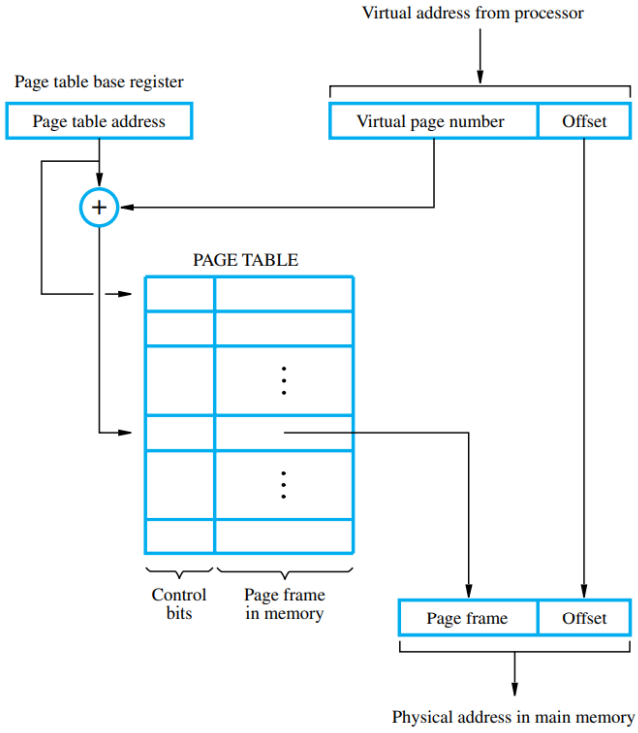
When a program doesn't have enough main memory to execute the programs virtual memory will allocate the extra memory to a secondary memory space, which will replace parts of the current main memory as the new ones are needed for execution. It's like caching secondary memory into main memory.

- **virtual or logical addresses:** binary addresses that the processor issues for either instructions or data.
- If a virtual address refers to a part of the program or data space that is currently in the physical memory, then the contents of the appropriate location in the main memory are accessed immediately. Otherwise, the contents of the referenced address must be brought into a suitable location in the memory before they can be used.
- **Memory Management Unit:** special hardware unit that keeps track of which parts of the virtual addres space are in the physical memory (main).
    - When the desired data or instructions are in the main memory, the MMU translates the virtual address into the corresponding physical address
    - If the data are not in the main memory, the MMU causes the operating system to transfer the data from the disk to the memory. Such transfers are performed using the DMA scheme that does not directly involve the processor



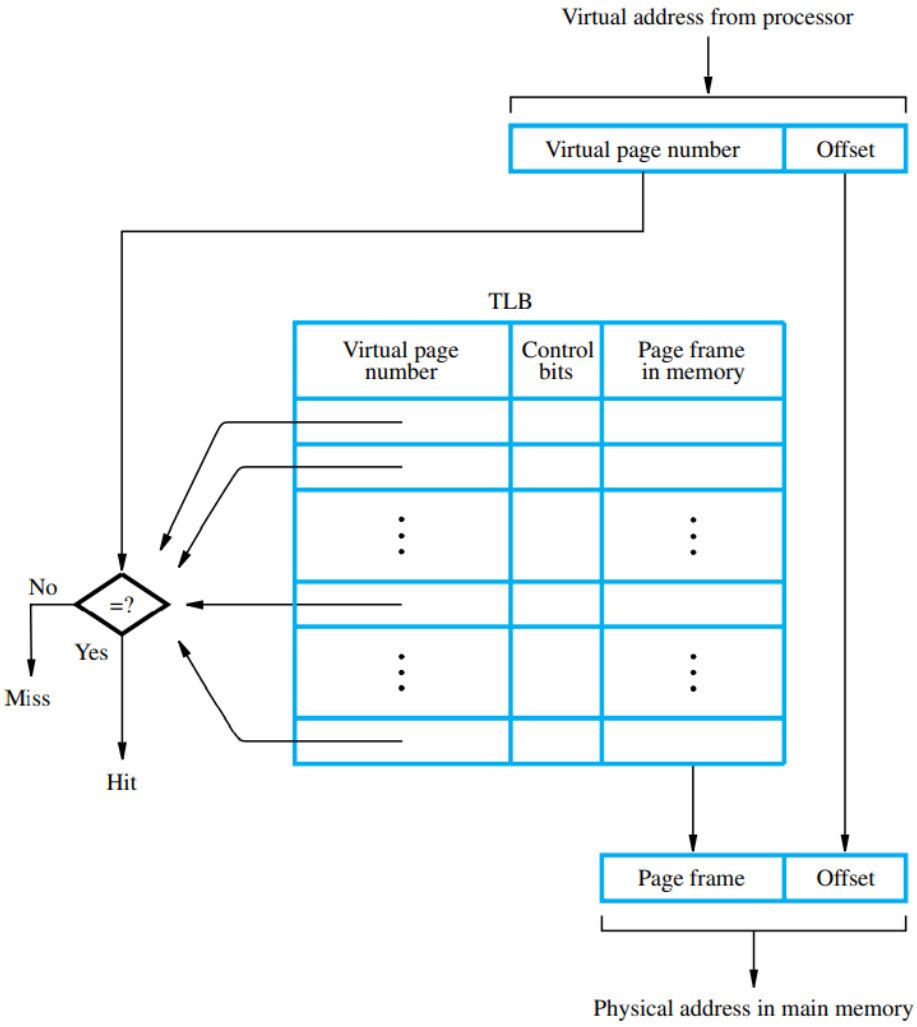
Virtual memory organization.

- **Address translation:** A simple method for translating virtual addresses into physical addresses is to assume that all programs and data are composed of fixed-length units called **pages.**



Virtual-memory address translation.

- The cache bridges the speed gap between the processor and the main memory and is implemented in hardware.
- The virtual-memory mechanism bridges the size and speed gaps between the main memory and secondary storage and is usually implemented in part by software techniques.
- **page table** This information includes the main memory address where the page is stored and the current status of the page.
- **virtual page number:** high-order bits
- **offset:** low-order bits (location of a particular byte or word within a page)
- **page frame:** An area in the main memory that can hold one page
- **page table base register:** Keeps the starting address of the page table
- **Translation Lookaside Buffer (TLB)**: Mantained within the MMU, the TLB functions as a cache for the page table in the main memory by containing the most recently accessed pages. In addition, it includes the virtual address of the page, which is needed to search the TLB for a particular page
- Address translation proceeds as follows:
    - Given a virtual address, the MMU looks in the TLB for the referenced page
    - If the page table entry for this page is found in the TLB, the physical address is obtained immediately
    - If there is a miss in the TLB, then the required entry is obtained from the page table in the main memory and the TLB is updated
    - It is essential to ensure that the contents of the TLB are always the same as the contents of page tables in the memory. When the operating system changes the contents of a page table, it must simultaneously invalidate the corresponding entries in the TLB



Use of an associative-mapped TLB.

- **page faults:** page that is not in the main memory, a page fault is said to have occurred. So like the cache miss
  - When it detects a page fault, the MMU asks the operating system to intervene by raising an exception (interrupt)
  - The operating system copies the requested page from the disk into the main memory
  - Concepts similar to the LRU replacement algorithm can be applied to page replacement
  - It is important to note that the write-through protocol, which is useful in the framework of cache memories, is not suitable for virtual memory. The access time of the disk is so long that it does not make sense to access it frequently to write small amounts of data.
- **system space:** separated from virtual space in which user application program resides dedicated for operatig system routines. Separate page table for each user program are arrranged. The physical main memory is thus shared by the active pages of the system space and several user spaces. However, only the pages that belong to one of these spaces are accessible at any given time.
  - **protection:** No program should be allowed to destroy either the data or instructions of other programs in the memory.
  - **supervisor mode:** The processor is usually placed in the supervisor mode when operating system routines are being executed and in the user mode to execute user programs.
  - **user mode:** some machine instructions cannot be executed. These are **privileged instructions**. They include instructions that modify the page table base register, which can only be executed while the processor is in the supervisor mode.
  - **shared pages:** Since a user program is executed in the user mode, it is prevented from accessing the page tables of other users or of the system space. Shared pages will therefore have entries in two different page tables

WEEK 7 – MEMORY LECTURE

DIRECT MEMORY ACCESS

Very simplistic co-processor with just 1 instruction, move.

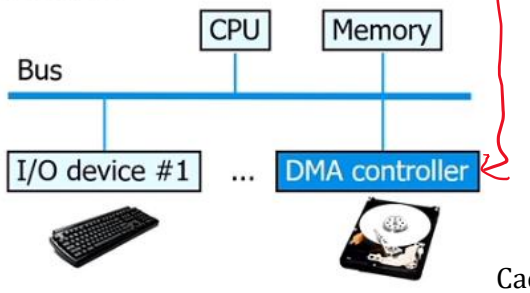
A separate active entity (**not** the CPU!) executes the I/O tasks
- user instructs the device about what to copy, where
- device takes care of data transport

- **Direct Memory Access**
  - 1 instruction: MOVE (= COPY)
- Special **I/O processors**
  - more versatile
  - run their own program

Parallel processing to the rescue!

- DMA controller
  - independent entity
  - specialized in data transfer
  - block based
- Bus priority
  - cycle stealing
  - burst mode

Q: what & how much is gained

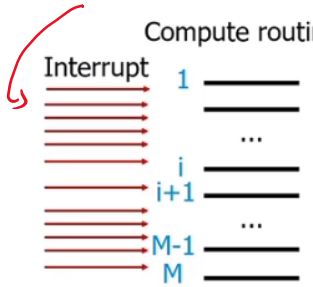


Double performance

Direct Memory Access

Why do we need it?

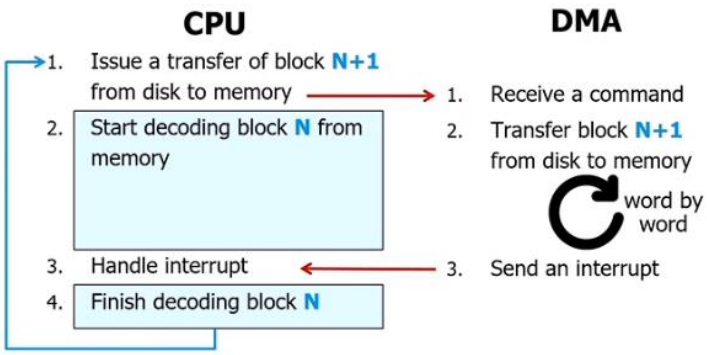
To avoid this



Cache will free up the bus from DMA bursts

CPU ↔ DMA interaction

Video-playback example



DMA fetches the next memory block while CPU is decoding the current one.

DMA controller

"Intelligent" I/O device

- Accessed as a normal I/O device
  - signals when DONE and/or by interrupt
- Acts as a bus master when transferring data
  - Sets addresses and controls lines

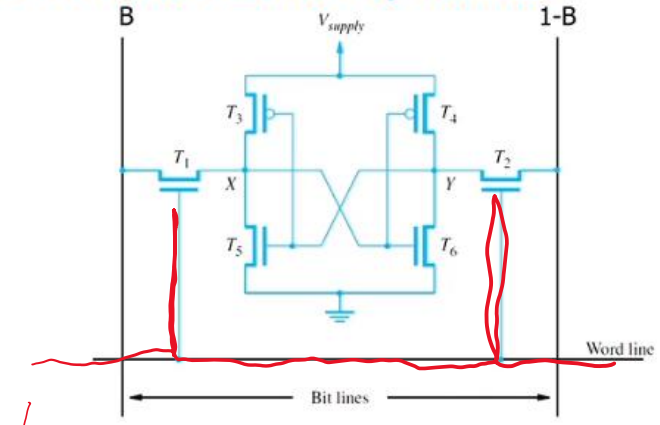
MEMORY ORGANIZATION

To shuffle larger words in one go

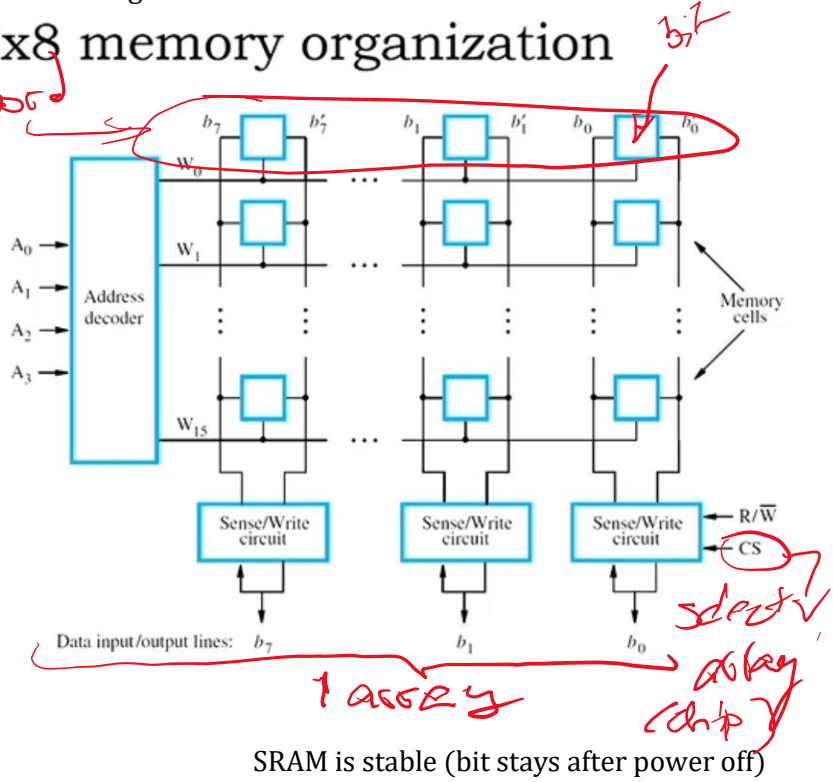
Usually, we
- keep memory aligned
- work with complete words

One bit in memory

Selected for read/write by word line

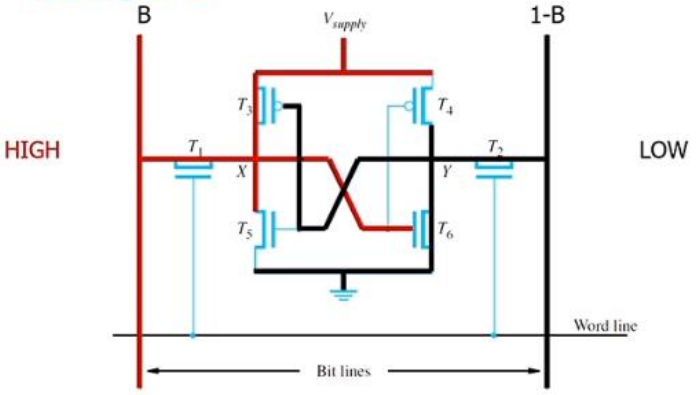


16x8 memory organization

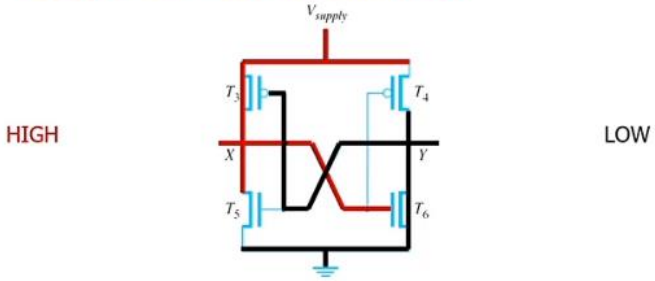


SRAM is stable (bit stays after power off)

Writing to B

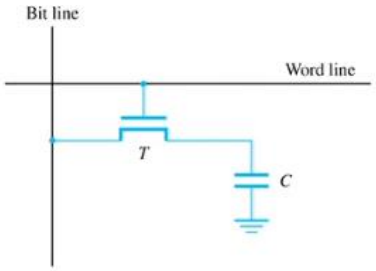


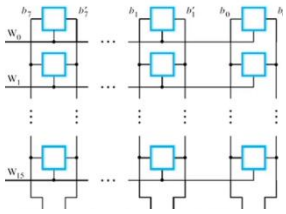
Keeps value when disconnected



- Without consuming power!
- Known as Static Random Access Memory (SRAM)

DRAM: Just a capacitor and a transistor which is cheaper than the 6 transistors of SRAM. Capacitor needs to be recharged (iff the state was 1). Charged capacitor means T = 1 (bit line = 1), empty capacitor T=0 (bit line = 0)

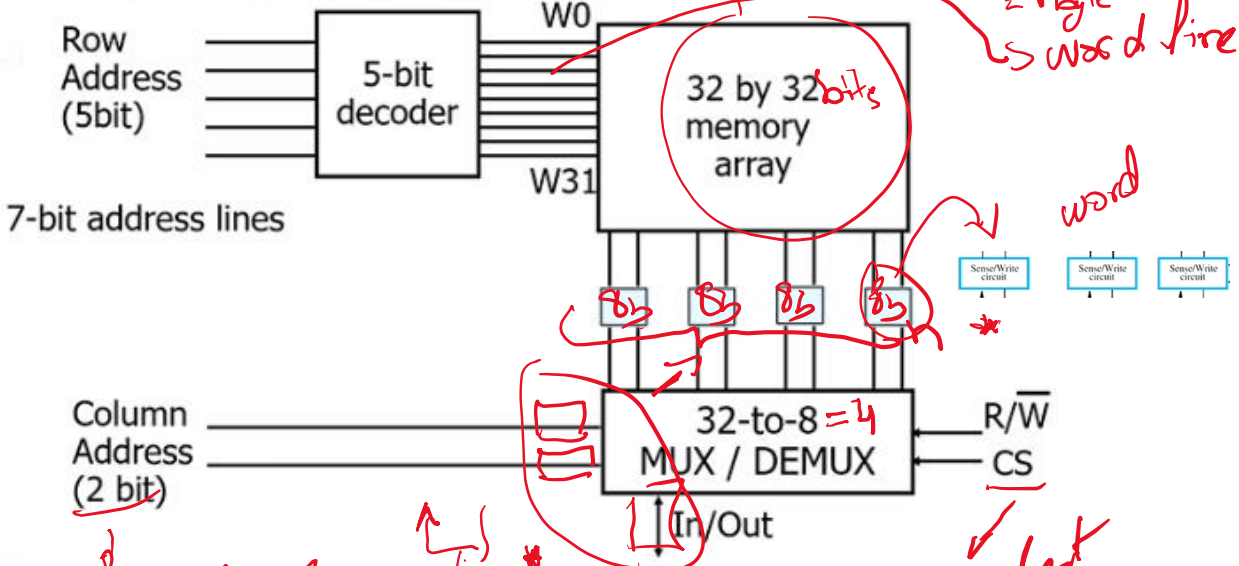




- Dynamic Random Access Memory (DRAM)
- Leakage current enforces periodic refresh

1024-bit memory organization 128x8

Splitting rows and columns



$\log_2(32/8) = 2$

7 bit byte addressable

$2^7 = 128$ bytes $= 128 \cdot 8$

$128 \times 8 =$ total

$\log_2(128) =$ address bits

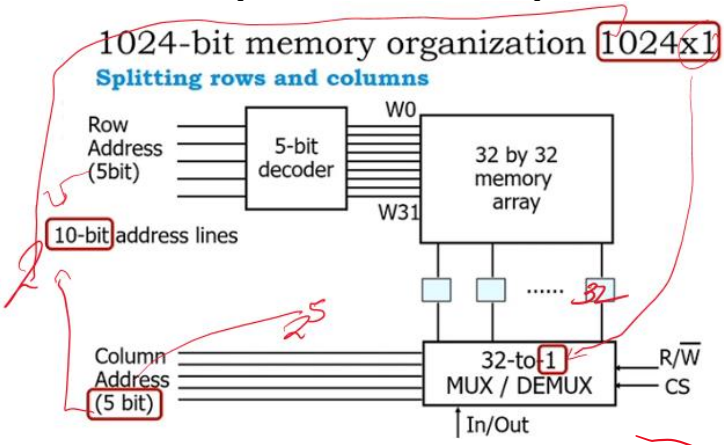
$\rightarrow \log_2(\frac{c}{8}) =$ Column address bits

$c =$ array colums size

$r =$ rows $= 2$ address row bits

only one 8 bit word is sent to the bus at a time

128x8 is more efficient than 1024x1 because you use words of 8 bits.
But 1024x1 is cheaper because it uses less pins



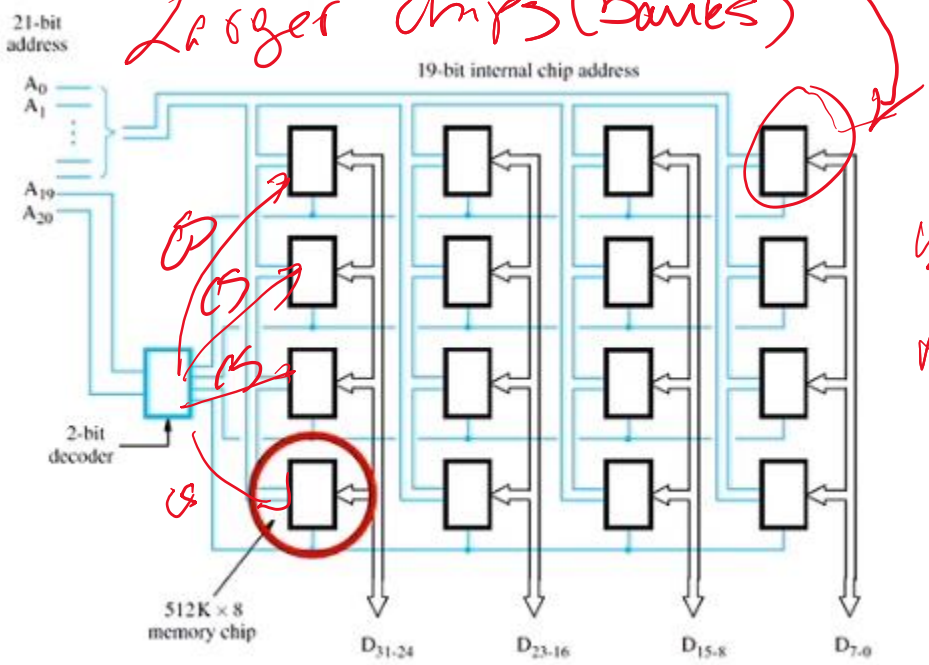
Cost expressed in #pins

128x8 organization	1024x1 organization
• 7 address pins	• 10 address pins
• 8 data pins	• 1 data pin
• 2 R/$\overline{W}$ + CS	• 2 R/$\overline{W}$ + CS
• 2 power + ground	• 2 power + ground
---- +	---- +
19 pins	15 pins

to refresh SDRAM bits

More speed is more expensive (more pins)

Larger chips (banks)

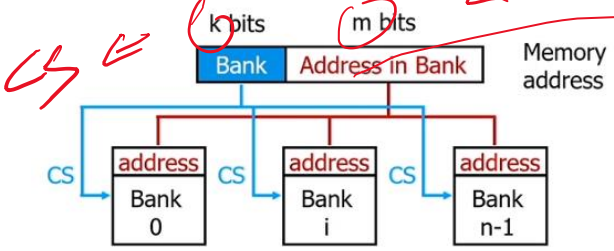


It's scalable because you reduce the risk of total failure.

same concept

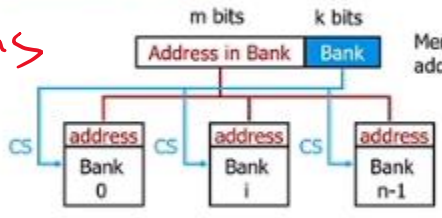
alternative

Memory banks
Flat layout



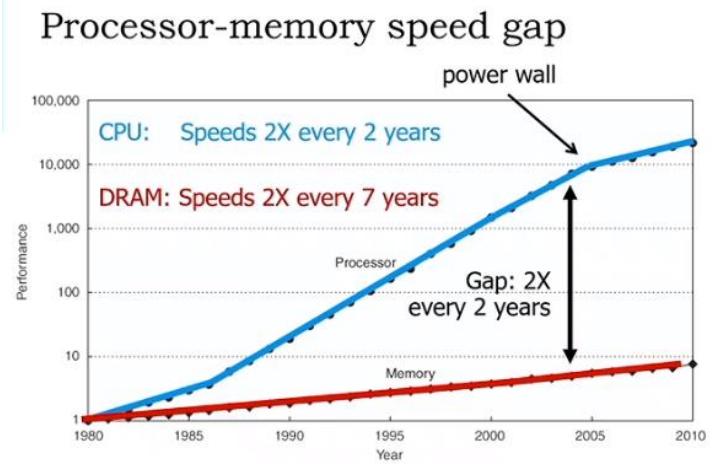
• Consecutive addresses served by the same memory

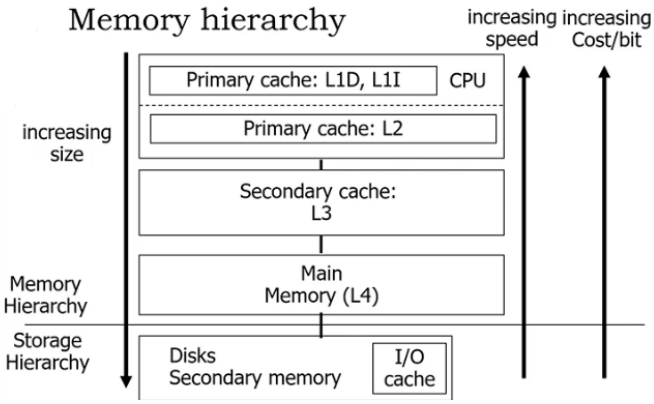
Memory banks
Interleaved layout



• Consecutive addresses served by consecutive banks
• Higher throughput (sequential access)

More banks available (expensive)



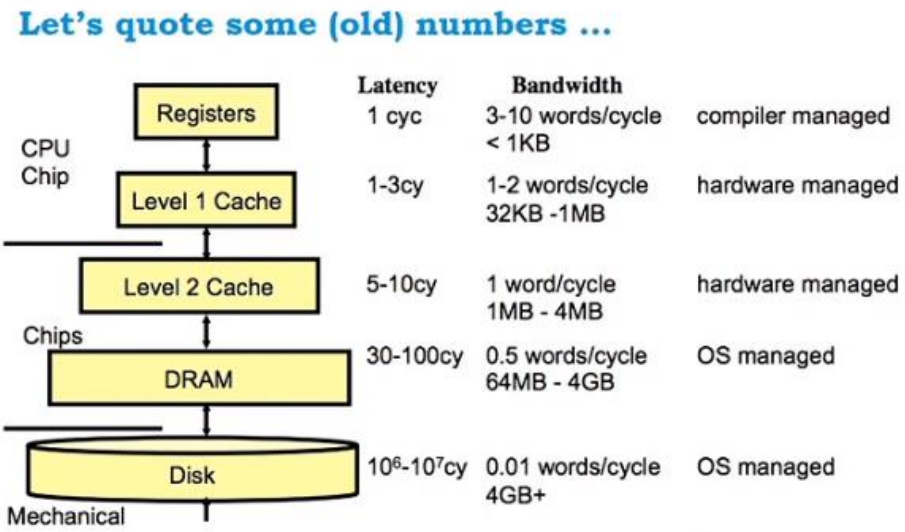


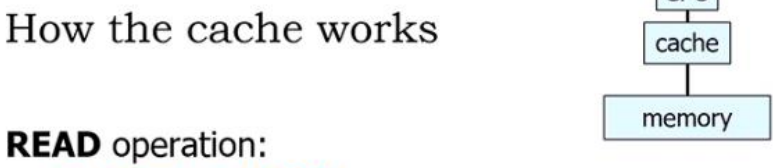
Power wall from too much generated heat in the circuits. Caches become more important as the gab between the CPU and the DRAM widens (Draining halt).

OS and hardware define which things go to which caches

Reason latency of the disk is 10^6 whereas the bandwidth is not that small is because the latency calculates the time (cpu cycle unit) of getting the first word wereas bandwidth looks at the average time of getting x words per cycle. Since the disk can process large words the average per word goes down.

OS decides how much DRAM (main memory) programs get and how much disk as well. OS can't decide caches.





READ operation:

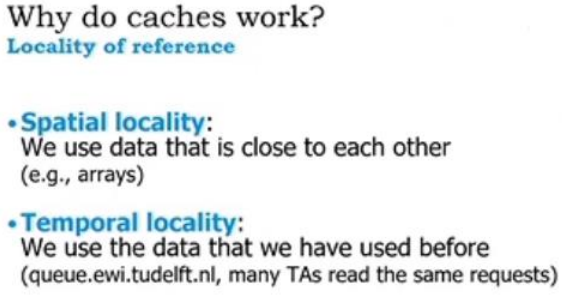
- if **not in cache (MISS)**, copy block into cache and read out of cache (possibly read-through)
- if **in cache (HIT)**, read out of cache

WRITE operation:

- if **not in cache (MISS)**, write in main memory
- if **in cache (HIT)**, write in cache, and either:
  - write in main memory (store through), or
  - set **modified** (dirty) bit, and write later

The cache is not only listening to the memory but also to the bus, so that if the disc (DMA) decicdes to change some words in memory the cache can be aware of it and either get updated or remove that cache (because it is invalid).

If cache writes something in cach only it will flag that word and only after the disc wants to access the equivalent memory location then will the cach interrupt that and write it on memory



Performance model for L1 cache

- Parameters:
  - cache hit ratio: **h**
  - access time cache: **c**
  - cache miss ratio: **1-h**
  - access time main memory: **m**
  - Average access time:

```
Avg access = c * h + (1 - h)(c + m)
           = c + (1 - h) * m
```

- These parameters depend on
  - application, possibly also on input (data-driven)
  - cache policy
  - ...

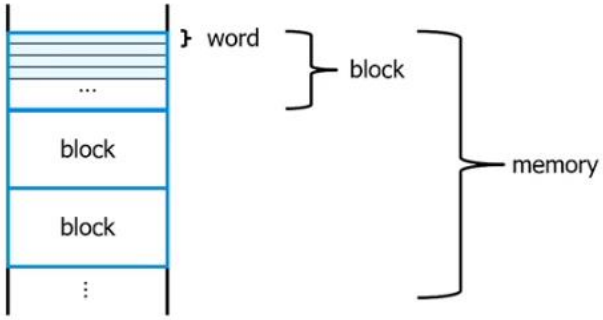
Design

- Transparent to
  - CPU (and programmer)
- Storage to hold
  - data
  - administration (in/out, clean/dirty)
- Block based
  - amortize admin overhead
  - efficient lookup (hit or miss?)
  - efficient memory access

We either read a block or write a block, we dont operate at individual (Memory) words

Memory blocks

- Caches access multiple memory words at a time



Fundamentals

Memory and cache are seen/structured as fixed-size blocks (aka cache lines)

- word size in bytes: **w**
- size of main memory in bytes: **$N = 2^n$**
- block size in words: **$b = 2^k$**
- number of blocks in main memory: **2^{n-k}** (= N / b)
- number of blocks in cache: **B**
- cache size: **B x b x w**
- Example: **n=16 (N=64KB), k=4 (b=16)**

How the cache works

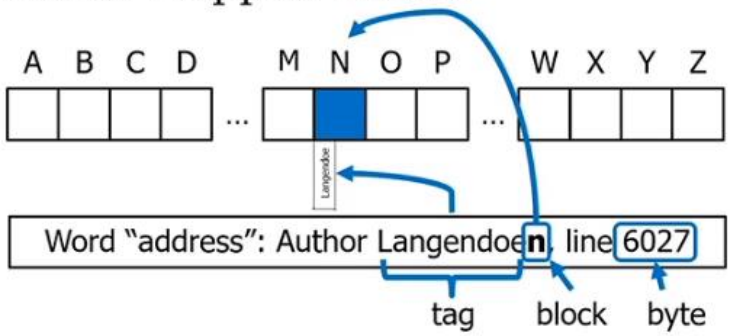
- Access word **w** in memory block **B**
- Is block **B** in the cache?
- **If NOT (MISS)**
  - get block **B** from memory
  - calculate where to place it in the cache
  - is position free?
  - if not, evict block at **B**'s place from the cache
  - write **B** to its cache location
- Access **w** in the cache

Where to place blocks in the cache?

- Cache is smaller than memory (by design)
  - multiple memory blocks map to the same cache block
- Mapping can be
  - static (direct mapped)
  - dynamic (fully associative)
  - mixed (x-way set associative)
- Cache performance (hit ratio) determined by
  - mapping strategy
  - locality of reference

Direct Mapped Cache

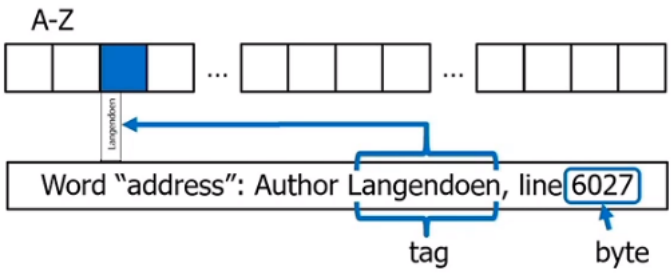
Cache 26 blocks large



- Main memory → Library
- Cache → Bookcase at home
- Block → Book
- Byte → Line

- Easy to look up books (blocks) in the bookcase (cache), so fast.
- Not flexible. Can lead to high **thrashing**: replacing blocks in cache that you still need. Books from authors 'Koe**n**' and 'Langendoe**n**' take the same place in the cache, so cannot be in the cache at the same time.

(Fully) Associative Cache



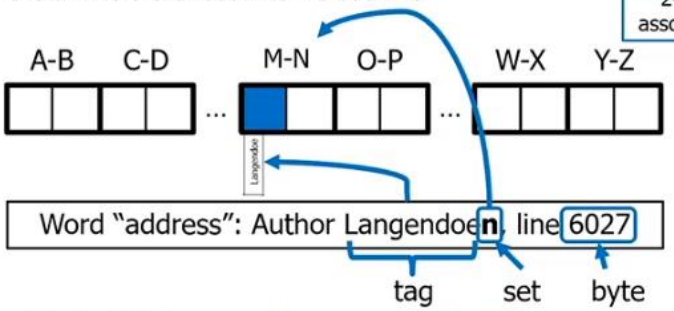
Everything goes to the same pile but you have to scan all the blocks to find them as there is no specific order other than pushing things.

Hardware still allows you to do searches in parallel at a not so high expense (just more gates).

- Difficult to look up books (blocks) in the bookcase (cache), so slow. We need to check **all** labels (in parallel).
- Very flexible. Can lead to high a **hit ratio**. Any book (block) can be anywhere in the bookcase (cache). Books from authors 'Koe**n**' and 'Langendoe**n**' can now be in the cache at the same time!

Set Associative Cache

Cache 26 blocks large
2-way associative

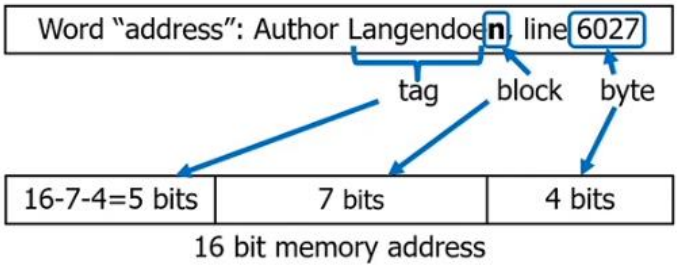


Combines Direct Mapped and Fully Associative. It allows for groups of sets where same set different tag elements can be store.

- Trade-off between **performance** and **flexibility**.
- Books from authors 'Koe**n**' and 'Langendoe**n**' can now be in the cache at the same time, but adding a third book from author 'Stefa**n**' requires one of the other two books to be evicted from the bookcase (cache).

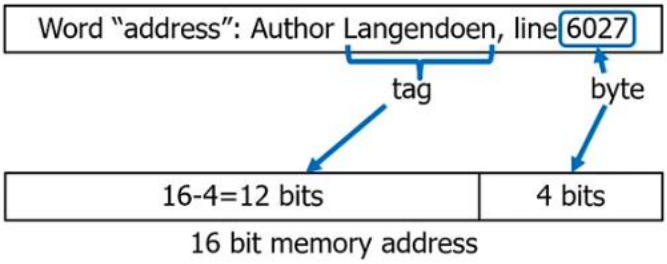
Back to reality… Direct Mapped

Example:
- 16 bit memory address
- 16 bytes in a block (4 bits to address)
- 128 blocks in cache (7 bits to address)



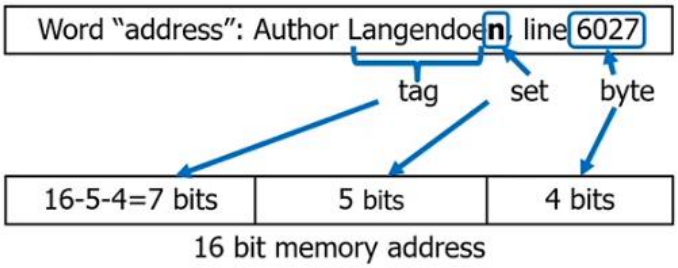
Back to reality… Fully Associative

Example:
- 16 bit memory address
- 16 bytes in a block (4 bits to address)
- 4K blocks in memory (12 bits to address)



Back to reality… Set Associative

Example:
- 16 bit memory address
- 16 bytes in a block (4 bits to address)
- 32 sets in cache (5 bits to address)

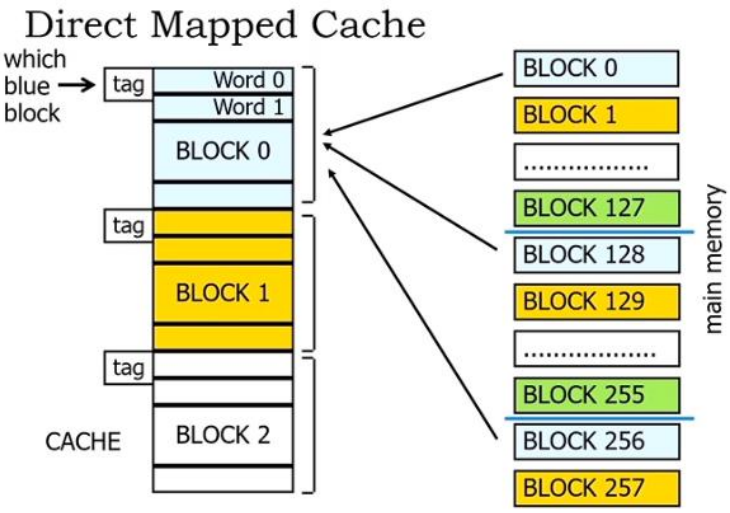


What happens if we take a x-way set-associative cache where:

1. x = 1
2. x = |blocks in cache|

x tells us how many blocks there are.
4-way set-associatve cache = 4 blocks per set
x = 1 means direct map cache (only 1 book per shelf, only 1 block per set)

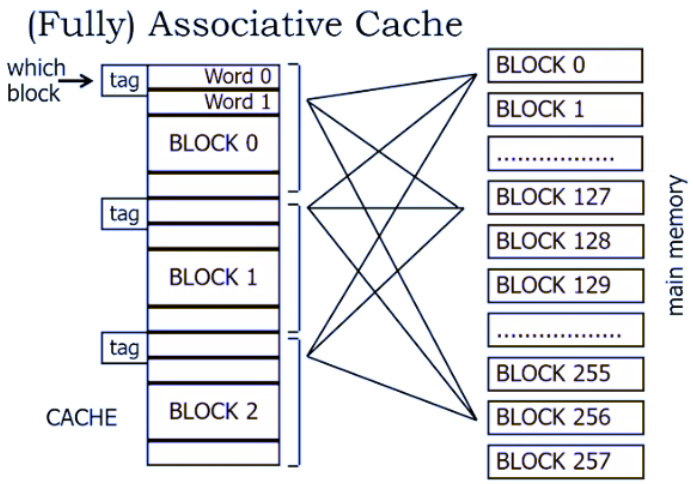
as many items in the set as many blocks in the cache means k*k = fully associative



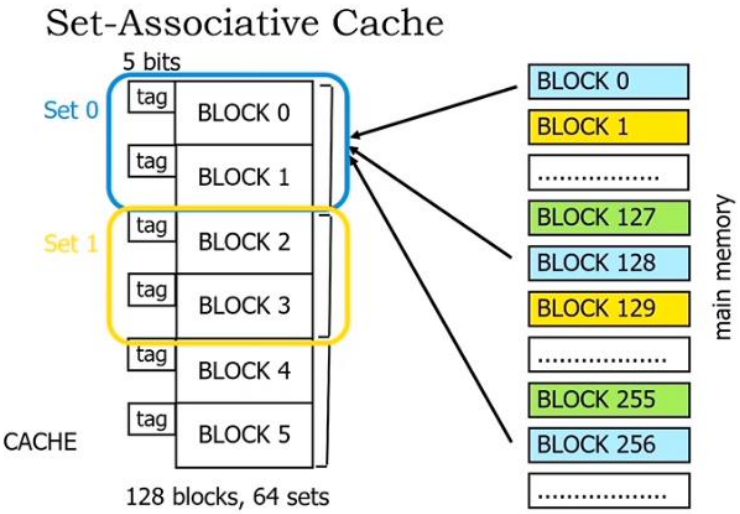
Transcript:

So we have colors for each block. Direct Mapped Cache takes the tag and the byte.
The bits in the middle (block), decide where to go in the cache

Con: If program wants to repeat using block 0 over and over, cache wont be able to store them all at the same time. It wil leither store 0 xor 128 xor 256



Anything can go anywhere (same collor)
As long as the number of blocks is not larger than the cache then we're fine



You can (opposite to direct maping) store more than 1 block in a set (2-way set)

CHAPTER 6 – PIPELINING

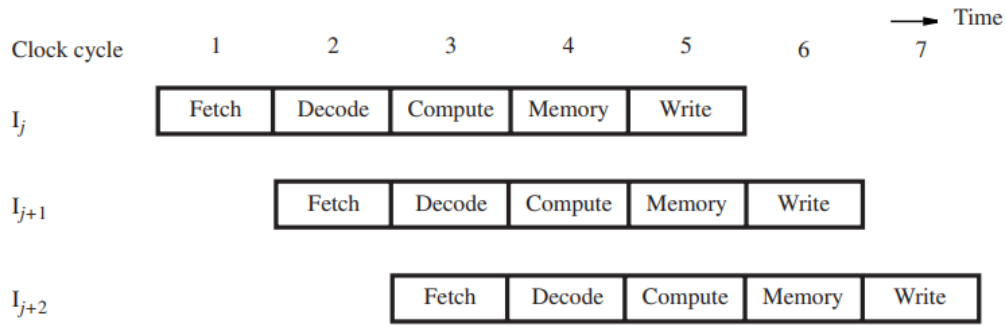
The five-stage processor of RISC and the corresponding datapath allow instructions to be fetched and executed one at a time. Therefore it takes five clock cycles to complete the execution of each instruction.

This could be pipelined so the fetch, decode, compute, memory and write stages can be done in parallel.

1. Instruction I_j is fetched in the first cycle and moves through the remaining stages
2. In the second cycle instruction I_{j+1} is fetched while $I_j$is on stage 2
3. In the third sycle instruction I_{j+2} is fetched while I_j is on stage 3 and I_{j+1} is on stage 2, and so forth.

Although any one instruction takes five cycles to complete its execution, instructions are completed, ideally at the rate of one per cycle (after the first 4). However, if the source register of I_{j+1} is the destination register of a memory writing operation of an instruction at I_j the operands of I_{j+1} won't be ready until stage 6, (opposite to the ideal scenario where they would be ready in stage 3). Which means I_{j+1} is **stalled** in the Decode stage for 3 cycles. Consquently I_{j+2} is also stalled and so forth.

- **hazard:** Any condition that causes the pipeline to stall

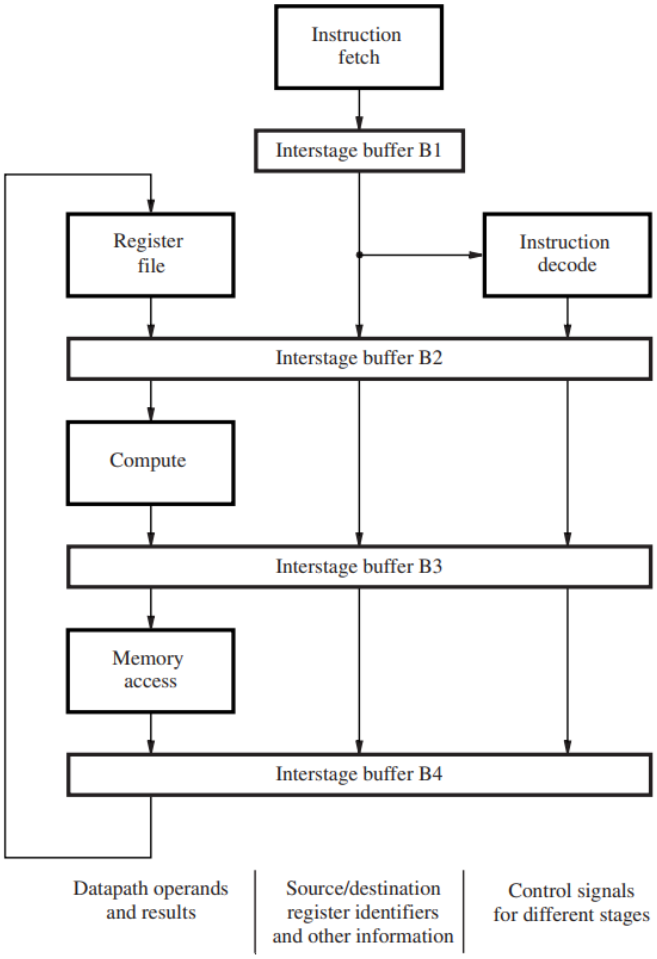


Pipelined execution—the ideal case.

Since register and things are being moved around at the same time it is necessary to save this information in **interstage buffers**. These include registers RA, RB, RM, RY and RZ

The interstage buffers are used as follows:

- Interstage buffer B1 feeds the Decode stage with a newly-fetched instruction
- Interstage buffer B2 feeds the Compute stage with the two operands read from the register file, the source/destination register identifiers, the immediate value derived from the instruction, the incremented PC value used as the return address for a subroutine call, and the settings of control signals determined by the instruction decoder.
- Interstage buffer B3 holds the result of the ALU operation, which may be data to be written into the register file or an address that feeds the Memory stage, and it also holds the incremented PC value passed from the previous stage, in case it is needed as the return address for a subroutine-call instruction
- Interstage buffer B4 feeds the Write stage with a value to be written into the register file. This value may be the ALU result from the Compute stage, the result of the Memory access stage, or the incremented PC value that is used as the return address for a subroutine-call instruction



A five-stage pipeline.

DATA DEPENDENCIES

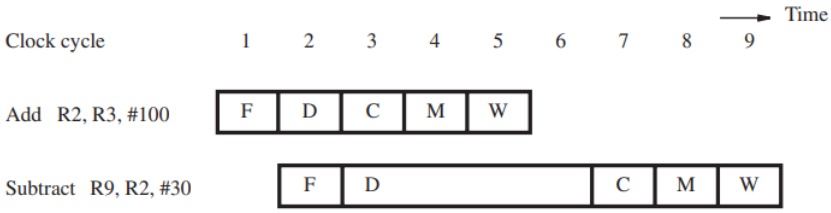


Figure 6.3 Pipeline stall due to data dependency.

Execution of:

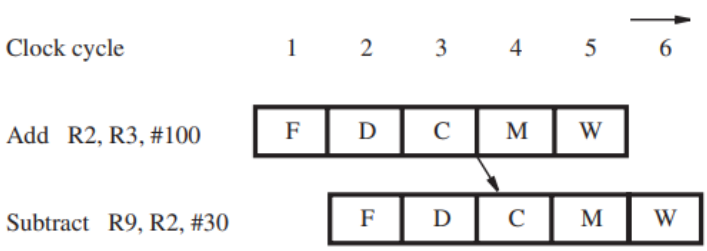
Add R2, R3 #100
Substract R9, R2, #30

data hazard introduced in the previous page. The Subtract instruction is stalled for three cycles to delay reading register R2 until cycle 6 when the new value becomes available.

- The control circuit must first recognize the data dependency when it decodes the Subtract instruction in cycle 3 by comparing its source register identifier from interstage buffer B1 with the destination register identifier of theAdd instruction that is held in interstage buffer B2.
- Subtract instruction must be held in interstage buffer B1 during cycles 3 to 5
- Add instruction proceeds through the remaining pipeline stages.
- control signals can be set in interstage buffer B2 for an implicit NOP (No-operation) instruction that does not modify the memory or the register file.
- Each NOP creates one clock cycle of idle time, called a **bubble.**

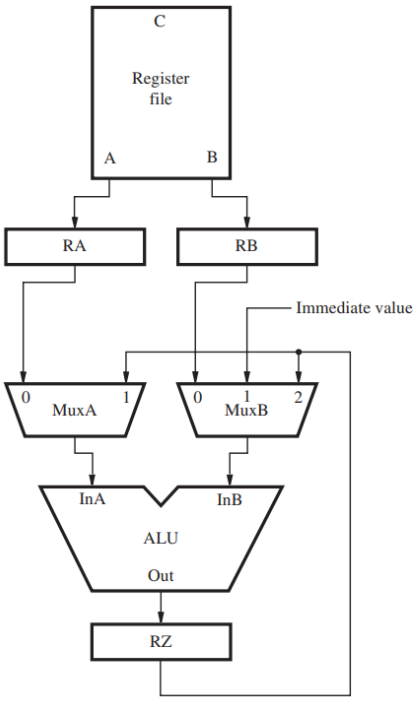
OPERAND FORWARDING

Pipeline stalls due to data dependencies can be alleviated through the use of operand forwarding. Considering the previous add and substract instructions.



Avoiding a stall by using operand forwarding.

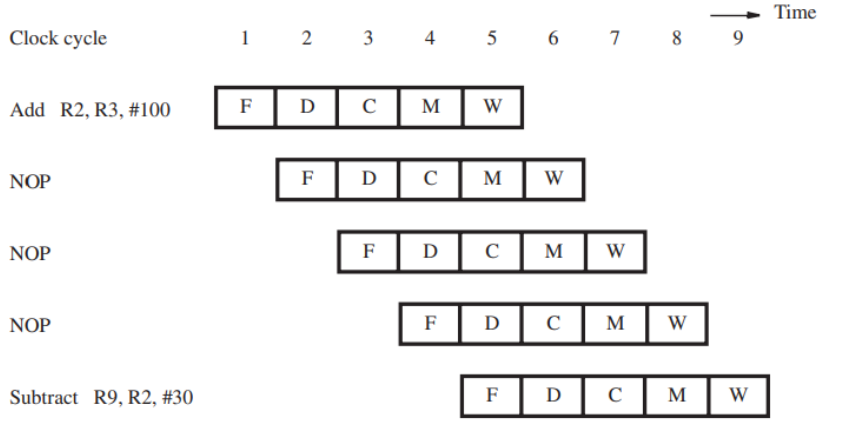
- Instead of substract to wait for stage 6 to decode the instruction with the register addresses, it could use the already available value computed at the end of stage 3.
- This value can be loaded into register RZ and rather than stall the Subtract instruction, the hardware can forward the value from register RZ to where it is needed in cycle 4



Modification of the datapath of Figure 5.8 to support data forwarding from register RZ to the ALU inputs.

A new multiplexer, MuxA, is inserted before input InA of the ALU, and the existing multiplexer MuxB is expanded with another input. The multiplexers select either a value read from the register file in the normal manner, or the value available in register RZ.

HANDLING DATA DEPENDENCIES IN SOFTWARE



Insertion of NOP instructions for a data dependency (done by the compiler) NOP takes 1 cycle, stalling it manually. This simplifies the hardware implementation at the expense of having larger code size. Execution time is still longer than operand forwarding

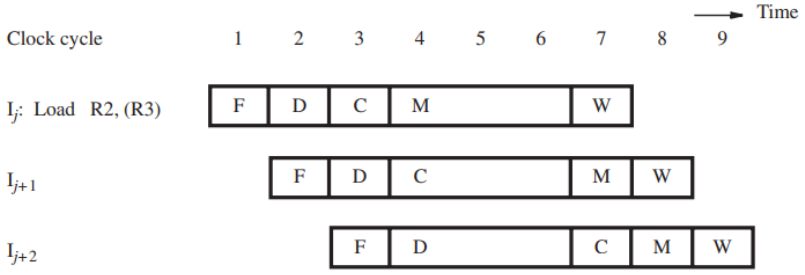
The compiler can attempt to optimize the code to improve performance and reduce the code size by reordering instructions to move useful instructions into the NOP slots

MEMORY DELAYS

A memory access may take ten or more cycles (3 in the figure for simplicity) a cache miss causes all subsequent instructions to be delayed. Consider:

```
Load        R2, (R3)
Substract   R9, R2, #30
```

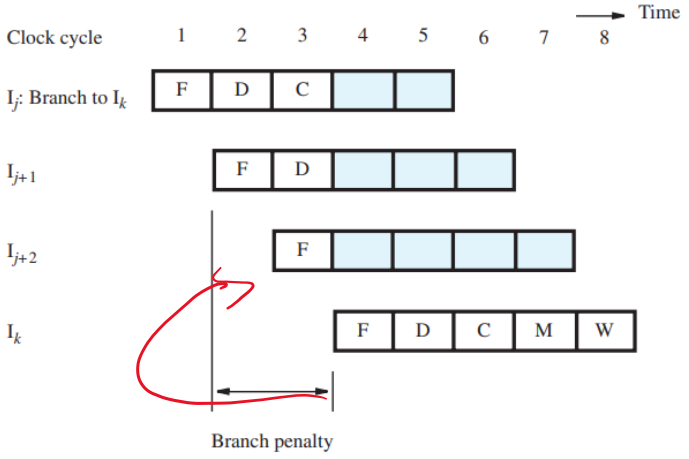
Operand forwarding cannot be done with memory as it takes more than one cycle to be fetched and wont be available until it is loaded into register RY in stage 5 (check BPU summary page 5 datapath structure).



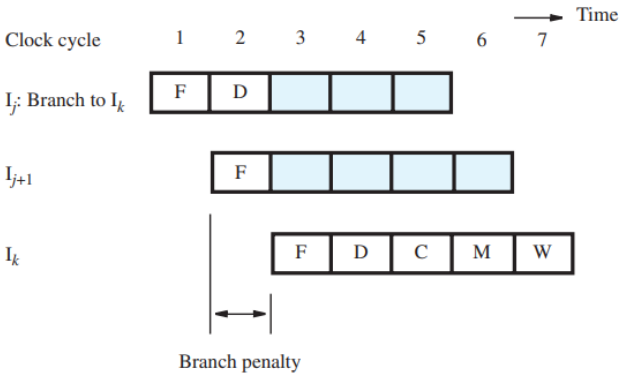
Stall caused by a memory access delay for a Load instruction.

The compiler can eliminate the one-cycle stall for this type of data dependency by reordering instructions to insert a useful instruction between the Load instruction and the instruction that depends on the data read from the memory. The inserted instruction fills the bubble that would otherwise be created. If a useful instruction cannot be found by the compiler, then the hardware introduces the one-cycle stall automatically. If the processor hardware does not deal with dependencies, then the compiler must insert an explicit NOP instruction.

BRANCH DELAYS



Branch penalty when the target address is determined in the Compute stage of the pipeline.



Branch penalty when the target address is determined in the Decode stage of the pipeline.

- **branch penalty:** Delay from branching

UNCONDITIONAL BRANCHES

- With a two-cycle branch penalty, the relatively high frequency of branch instructions could increase the execution time for a program by as much as 40 percent.
- Reducing the branch penalty requires the branch target address to be computed earlier in the pipeline. Rather than wait until the Compute stage.
- it is possible to determine the target address and update the program counter in the Decode stage
- Thus, instruction Ik can be fetched one clock cycle earlier, reducing the branch penalty to one cycle.
- A second adder is needed in the Decode stage to compute a branch target address for every instruction
- When the instruction decoder determines that the instruction is indeed a branch instruction, the computed target address will be available before the end of the cycle. It can then be used to fetch the target instruction in the next cycle.

CONDITIONAL BRANCHES

Branch_if_[R5]=[R6] LOOP

- The result of the comparison in the third step determines whether the branch is taken.
- The branch condition must be tested as early as possible to limit the branch penalty (he comparator that tests the branch condition can also be moved to the Decode stage).

- Moving the branch decision to the Decode stage ensures a common branch penalty of only one cycle for all branch instructions

THE BRANCH DELAY SLOT

```
        Add              R7, R8, R9
        Branch_if_[R3]=0 TARGET
        I_{j+1}
        ⋮
TARGET: I_k
```

(a) Original sequence of instructions containing a conditional branch instruction

```
        Branch_if_[R3]=0 TARGET
        Add              R7, R8, R9
        I_{j+1}
        ⋮
TARGET: I_k
```

b) Placing the Add instruction in the branch delay slot where it is always executed

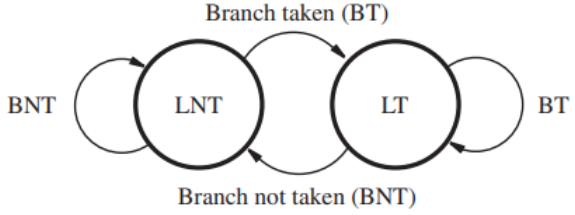
Filling the branch delay slot with a useful instruction.

- In all cases, the instruction immediately following the branch instruction is always fetched.
- **delayed branching**: To reduce branch penalty the branch delay slot technique attempts to find a suitable instruction to occupy the delay slot of the branch instruction, one that needs to be executed even when the branch is taken.
- It can do so by moving one of the instructions preceding the branch instruction to the delay slot (as long as data dependencies are preserved).
- If a useful instruction is found, then there will be no branch penalty.
- If no useful instruction can be placed in the delay slot because of constraints arising from data dependencies, a NOP must be placed there instead.
- The effectiveness of delayed branching depends on how often the compiler can reorder instructions to usefully fill the delay slot (70%).

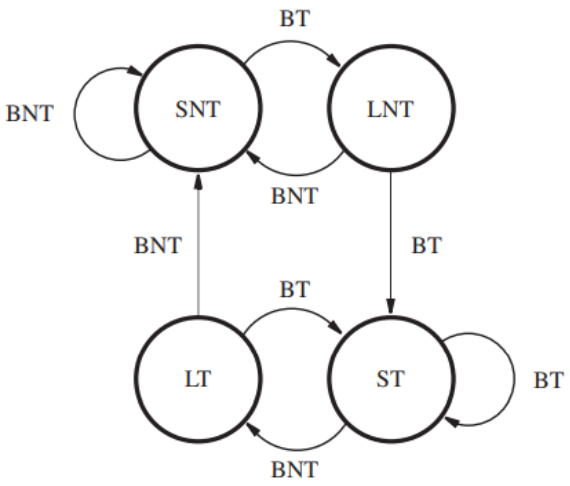
BRANCH PREDICTION

- Making the branch decision in cycle 2 of the execution of a branch instruction reduces the branch penalty.
- the instruction immediately following the branch instruction is still fetched in cycle 2 and may have to be discarded.
- decision to fetch this instruction is actually made in cycle 1, when the PC is incremented while the branch instruction itself is being fetched
- to reduce the branch penalty further, the processor can anticipate an instruction being fetched is a branch instruction and **predict** its outcome to determine which instruction should be fetched in the next cycle.
- **Static branch prediction:** Assume that the branch will not be taken and fetch the next instruction in sequential order. There will only be penalty when the prediction is incorrect. Assuming randomness, this gives 50% accuracy. However backward branches at the end of a loop are taken most of the time, for such a loop is better to assume that the branch is gonna be taken. The processor can determine the static prediction by checknig the sign of the branch offset. Alternatively, the **machine encoding of a branch instruction may include a bit that indicates how to predict the instruction**.
- **Dynamic Branch Prediction:** The processor hardware keeps track of branch history to make better predections. Simplest form is to use the last result as prediction. **Works well inside program loops**. The track history can be expanded to more than just the last one, i.e. 4-state algorithm vs 2-state algorithm.

BT - Branch taken
BNT - Branch not taken
ST - Strongly likely to be taken
LT - Likely to be taken
LNT - Likely not to be taken
SNT - Strongly likely not to be taken



(a) A 2-state algorithm



(b) A 4-state algorithm

- **Branch Target Buffer:** small fast memory that contains the extra information that the processor needs to keep for dynamic branch prediction. The branch target buffer contains a lookup table for each branch with:
  - the address of the branch instruction.
  - one or two state bits for the branch prediction algorithm
  - the branch target address
- the table has a limited size (1024ish entries), containing information for only the most recently executed branch instructions
- **(not necessarily branch) speculative execution:** subsequent instructions based on an unconfirmed prediction are fetched, dispatched, and possibly executed, but are labeled as being speculative so that they and their results may be discarded if the prediction is incorrect
- **reservation stations**: buffer for sepeculative execution, they hold information and operands relevant to each dispatched instruction

PERFORMANCE EVALUATION

- **non-pipelined processor - basic performance equation:**
  - T: execution time
  - N: Dynamic Instruction Count
  - S: Average number of clock cycles to fetch and execute one instruction
  - R: clock rate in cycles per second

$$\boldsymbol{T = \frac{N * S}{R}}$$

- **instruction throughput (non-pipelined):** Number of instructions executed per second

$$P_{np} = \frac{R}{S}$$

RISC, when there are no cache misses, uses 5 cycles to execute all instructions. So S = 5
Pipelining improves performance by overlapping the execution of successive instructions

- **instruction throughput (platonic pipelined):** In the abscence of stalls

$$P_p = R$$

Remember that there are millions of instructions so the first 4 that are not 100% overlapped are insignificant. So a n-stage pipeline can potentially increase the throughput by a factor of n. In reality there are diminishing returns but recent processor implementations ahve 20 stages with clock rates of several GHz

DELAYS

The operations with the longest delay dictate the cycle time, and hence the clock rate R.

- δ_{stall}: increased difference from S, where S = 1 in an ideal world.
  δ_{stall} = Dynamic Instructions % * Dependent Instructions (of the dynamic) % * 1

- **instruction throughput pipelined (with stalls):**

$$P_p = \frac{R}{1 + \delta_{stall}}$$

- The compiler can improve performance by reducing the number of times that a Load instruction is immediately followed by a dependent instruction.
- A stall is eliminated each time the compiler can safely move a nearby instruction to a position between the Load instruction and the dependent instruction
- **mispredicting branches:** Assume target address are determined in the Decode stage of the pipeline

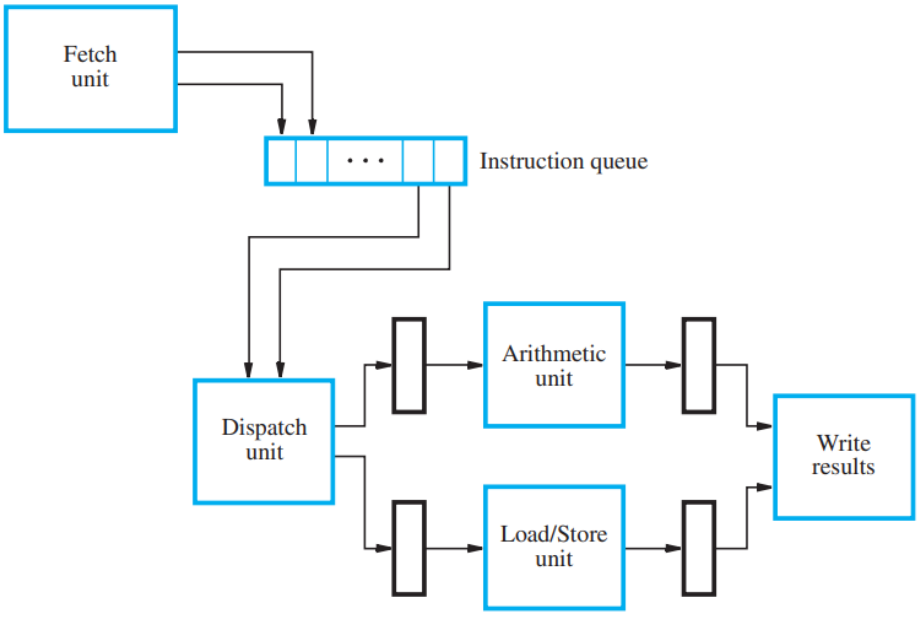
$$\delta_{branch\_penalty} = standard\ branch\ penalty\ (1)\ *\ branch\ \%\ *\ average\ prediction$$

- **cache misses:**

$$\delta_{miss} = (m_i * d * m_d) * p_m$$

Where p_m = stalled pipeline cycles from cache misses; m_i = instruction miss %; d = load and store instructions % (that involve the cache) and m_d = operand miss %

SUPERSCALAR OPERATION



A superscalar processor with two execution units.

The maximum throughput of a pipelined processor is one instruction per clock cycle. **Superscalar** procesor can acheive more than 1 instruction per clock cycle by equipping the processor with "**multiple-issue**" multiple execution units (each could be also pipelined).

- a superscalar processor has a more elaborate **fetch unit** that fetches two or more instructions per cycle before they are needed and places them in an instruction queue
- dispatch unit: takes two or more instructions from the front of the queue, decodes them, and sends them to the appropriate execution units.
- At the end of the pipeline, another unit is responsible for writing results into the register file.

- It incorporates two execution units, one for arithmetic instructions and another for Load and Store instructions.
- An arithmetic instruction and a Load or Store instruction must obtain all their operands from the register file when they are dispatched in the same cycle to the two execution units. **The register file must now have four output ports instead of the two output ports** needed in the simple pipeline.
- an arithmetic instruction and a Load instruction must write their results into the register file when they complete in the same cycle. Thus, the register file must now have **two input ports instead of the single input port for the simple pipeline**.
- the register file allows two results to be written in the same cycle because the destination registers are different.
- Otherwise, one instruction is stalled to ensure that results are written into the destination register in the same order as in the original instruction sequence of the program
- As long as such dependencies are handled correctly, there is no reason to delay the execution of an unrelated instruction. If there is no dependency between a pair of instructions, the order in which execution is completed does not matter. However **exceptions** will lead to a processor executing the second instruction, which may have had relied on the first one not having an exception, this is called an **impricese exceptions**.
- **precise exceptions:** delaying or buffering instructions to give room for exceptions at the expense of more complex hardware
- **comittment step:** moving the temporary registers to the permanent ones. The effect of an instruction cannot be reserved after this point.
- **register renaming:** a temporary register takes the role of a permanent register during a period of time. There may be as many temporary registers as there are permanent registers.
- **commitment unit:** sepcial control unit that uses a separate queue called the **reorder buffer** to determine which instruction(s) should be committed next (if out of order execution is allowed), this guarantees in-order commitment. Instructions are **retired** after the temporary registers have been moved to the fixed ones.
- **dispatch order operations:** the dispatch unit must ensure that all the resources needed for the execution of an instruction are available.
- **deadlock:** a situation that can arise when two units, A and B, use a shared resource and both of them are waiting for the other in a vicious circle so that neither can complete its execution

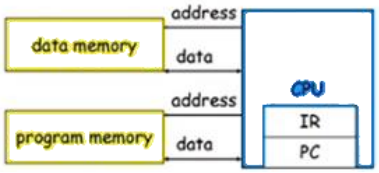
WEEK 8 – PIPELINING LECTURE

Remember: without caching there's no possible DMA

Boosting cache performance

Harvard architecture

- Exploit difference in locality
  - instructions – spatial + temporal
  - data – temporal + spatial



- Separate data and instruction paths

- Harvard architecture treats data and instructions differently.
- This structure allows to fetch data and instructions at the same time
- It also allows us to use different cache mappings for data and program memory.
- Programs are typically sequential, so they respond better to direct mapping.
- Data is all over the place, responds better to associative case.

Performance

Latency (L)
- the time for one instruction to finish
- lower is better

Throughput (T)
- the #instructions per time unit
- higher is better
- the time per instruction decreases, but **only on average**

Pipelining aims to increase the throughput. Which is optimal for general purpouse computing (rather than dedicated circuitry).

Improving CPU performance

- Increase clock frequency → **latency** boost
  - problem: power wall
- Multiple threads & cores → **throughput** boost
  - problem: parallel programming
- Pipeline execution → **throughput** boost
  - problem: limited effect
- Dedicated circuitry → **latency** boost
  - problem: one-trick pony

Pipelining

Basic idea

Time →

CPU	**Fetch** Instruction 1	**Execute** Instruction 1	**Fetch** Instruction 2	**Execute** Instruction 2

Time →

Fetch	**Fetch** Instruction 1	**Fetch** Instruction 2	**Fetch** Instruction 3	**Fetch** Instruction 4
CPU				
Execute		**Execute** Instruction 1	**Execute** Instruction 2	**Execute** Instruction 3

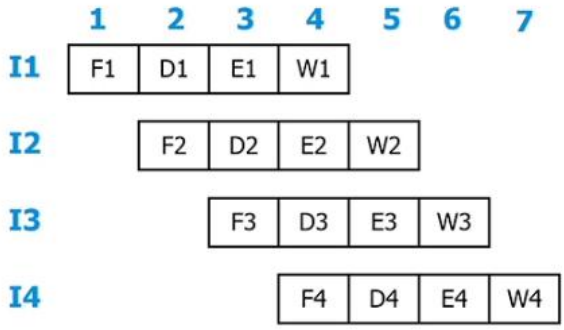
Instruction stages

- **F**etch instruction
- **D**ecode instruction and fetch operands
- **E**xecute operation
- **W**rite result

Note: this is a simplified model with only 4 stages.
The textbook presents a model with 5 stages.
The pipelining principle is the same.

Pipelined execution

1 stage / cycle

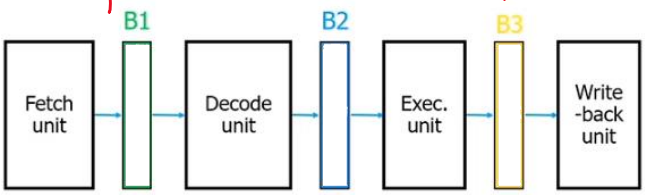


We need to modify our hardware from the previous BPU architecture as it was not designed to do these stages in parallel. So we need **buffers**.

Hardware organization

CPU internals

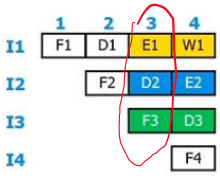
- Insert buffers to allow pipeline stages to act **independently**



Buffer content

At start of cycle 4

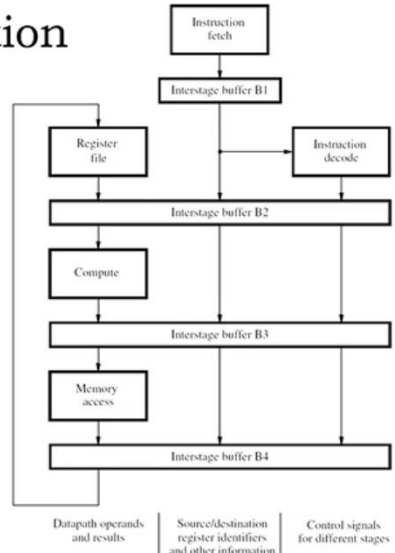
- **B1 (F -> D):**
  - instruction I3
- **B2 (D -> E):**
  - the source operands of I2
  - the operation
- **B3 (E -> W):**
  - the result of the execution of I1



Hardware organization

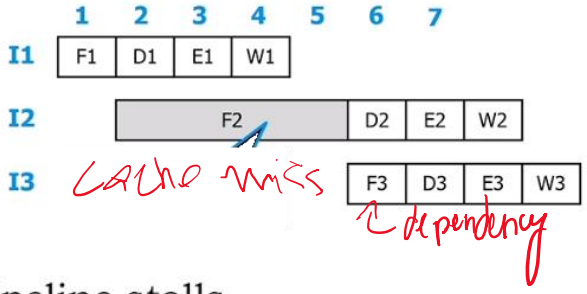
Example: RISC machine

Pipeline logic
- data path
- operands specifiers
- control signals



There is a trade-off between aiming for a high throughput with lots of stages and increasing the risk of longer pipeline stalls.

Pipeline stalls

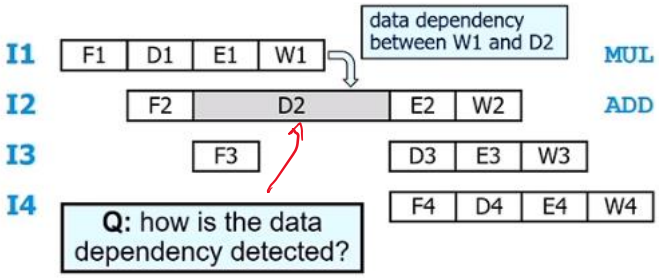


Pipeline stalls
Hazards (causes)

- Cache miss [fetch, decode]
- Dependency between instructions [decode]
  - e.g., data output of one instruction is input for next
- Branching [exec]
- Long operation [exec]
  - e.g., division

Pipeline stalls
Data dependencies

```
MUL  R2,R3,R4    /* R4 destination */
ADD  R5,R4,R6    /* R4 source */
```

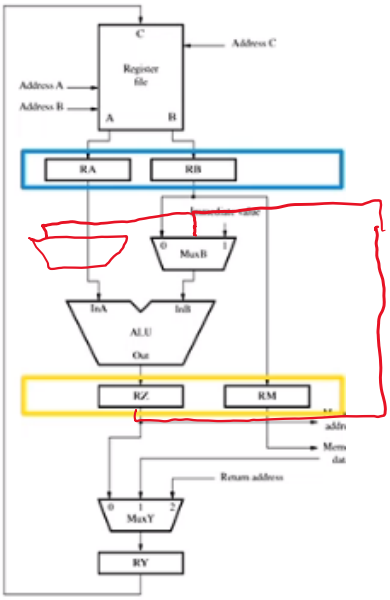


In the decoding stage you can actually see the data dependency between instruction I2 and instruction I1.

Solution: The output of the current instruction is ready with **forwarding** in the next stage so that we can execute instructions back to back instead of waiting for the write step.

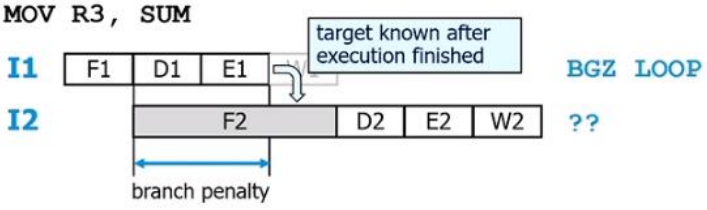
Data forwarding
HW to the rescue

- Add fast path from ALU output to input



Pipeline stalls
Branching

```
BGZ  LOOP
MOV  R3, SUM
```



- For **unconditional** branches
  - target known after decoding
  - branch penalty reduces to 1 cycle

unconditional: It the branch is just a jump instruction, there's no need for the ALU/comparator to execute anything, so the result is known right after decoding (just 1 bubble)

conditional prediction: Instead of going to the ALU/comparator, we could also assume a-priori the result of the loop and jump (or not) directly.

Branch delay slot

SW to the rescue

```
ADD R6,R7,R8
BGZ R3 LOOP
MOV R8, SUM
```

- Penalty = 1 cycle
- Use idle slot

branching occurs often: **20%** of instruction count

```
BGZ R3 LOOP
ADD R6,R7,R8
MOV R8, SUM
```

branch delay slot: **always** executed **before** the branch effects

BGZ LOOP: PC = PC + offset

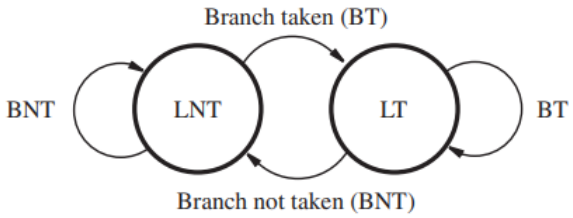
offset = body lines of the loop

Branch prediction

Statistics matter

- Static prediction
  - backwards branches likely to be taken (loops)
  - forward branches mostly skipped (rule vs. exception)
  - sign bit (or compiler directed)
- Dynamic prediction
  - maintain info per branch instr. (branch target buffer)
  - do as before

finite state machine ->



(a) A 2-state algorithm

For long operations such as divisions since they appear rarely the don't require our attention as they don't have an impact on performance.

Performance effects

A simple model

- Execution time of a program: **T**
- Instruction count: **N**
- #cycles per instruction, CPI: **S** (e.g., 4)
- Clock rate: **R** (e.g., 1 GHz)

Without pipelining: $T = (N \times S) / R$

With an **n**-stage pipeline: $T' = T / n$

CHAPTER 12 - PARALLEL PROCESSING AND PERFORMANCE

HARDWARE MULTITHREADING

- Operating system (OS) software enables multitasking of different programs in the same processor by performing context switches among programs
- Processes (any information that describes the current state of the program execution) may be associated with applications such as Web-browsing, word-processing, and music-playing programs that a user has opened in a computer. **Each process has a corresponding thread**.
- it is possible for multiple threads to execute portions of one program and run in parallel as if they correspond to separate programs. But all threads that are part of a single program run in the same address space and are associated with the same process.
- **hardware multithreading:** To deal with multiple threads efficiently, **a processor is implemented with several identical sets of registers**, including multiple program counters.
- The state of the previously active thread is preserved in its own set of registers.
- **coarse-grained multithreading:** an about to stall processor quickly switches to a different thread and continue to fetch and execute other instruction.

- **fine-grained orinterleavedmultithreading:** switch after every instruction is fetched. Throughput may be increased by interleaving instructions from many threads, but it takes longer for a given thread to complete all of its instructions.

VECTOR (SIMD) PROCESSING

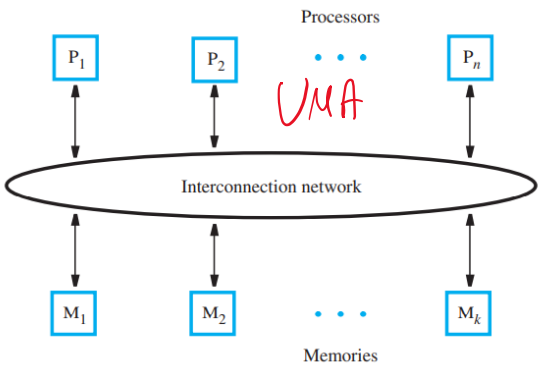
- **vector:** array of elements usch as integers or floating-point numbers.
- **vector instructions / single-instruction multiple-data (SIMD) instructions:** A processor can be enhanced with multiple ALUs. In such a processor, it is possible to operate on multiple data elements in parallel using a single instruction. Can only be used when the operations performed in parallel are independent. This is known as **data parallelism.**
- **vector registers:** they can hold several data elements. **L = vector length** = number of data elements = number of operations that can be performed in parallel with multiple ALUs
- VectorAdd.S Vi, Vj, Vk: just a vector sum that takes vector registers operands and saves it in a vector register
- storing and loading vectors just places elelements consecutively in the destination and read consequitive elements into a vector.
- **vectorizable:** such as high-level integer arrays. Where operations for all elements of the array can be done in parallel
- Vectorizable loops exist in programs for applications such as computer graphics and digital signal processing.

GRAPHICS PROCESSING UNITS (GPUS)

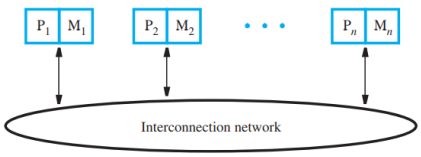
- The primary purpose of GPUs is to **accelerate the large number of floating-point calculations** needed in high-resolution three-dimensional graphics, such as in video games
- operations involved in these calculations are often **independent**
- a large GPU chip contains hundreds of simple cores with floating-point ALUs to perform them in parallel
- A GPU chip and a dedicated memory for it are included on a video card
- A small program is written for the processing cores in the GPU chip
- A large number of cores execute this program in parallel
- The cores execute the same instructions on parallel, but operate on different data elements.
- Before initiating the GPU computation, the program in the host computer must first transfer the data needed by the GPU program from the main memory into the dedicated GPU memory
- After the computation is completed, the resulting output data in the dedicated memory are transferred back to the main memory
- There's a C extension to deal with NVIDIA's GPU so that an entire program can be written in C.
    - The compiler will partion the final object into machine instructions for the GPU and CPU
    - An open standard called OpenCL has been proposed by industry as a programming framework for systems that include GPU chips from any vendor

SHARED MEMORY MULTIPROCESSORS

- Implementing a large memory in a single module would create a bottleneck when many processors make requests to access the memory simultaneously
- It can be alleviated by distributing memory across multiple modules so that simultaneous requests from different processors are more likely to access different memory modules
- An interconnection network enables any processor to access any module that is a part of the shared memory
- **(UMA) Uniform Memory Access multiprocessor:** A system which has the same network latency for all accesses from the processors to the memory modules.
- **(NUMA) Non-Uniform Memory Access multiprocessors:** For better performance, they place a memory module close to each processor, resulting in a collection of nodes that have different latencies.



NUMA →

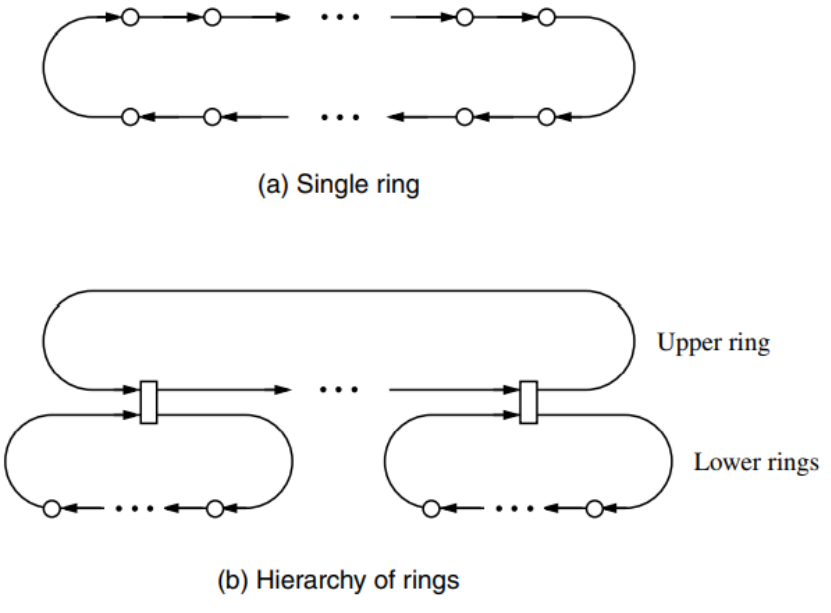


INTERCONNECTION NETWORKS

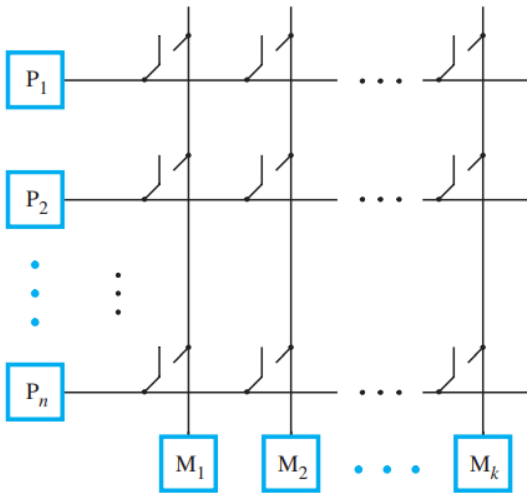
- The interconnection network must allow information transfer between any pair of nodes in the system
- The traffic in the network consists of requests (such as read and write) and data transfers
- **bandwidth:** capacity of a transmission link to transfer data bytes per second.
- **effective throughput:** rate of data transfer, which is less than the available bandwidth because a link must carry information that coordinates the transfer of data.
- **packets:** information transfers through the network, of a fixed length and specified format
- Ideally, a complete packet would be handled in parallel in one clock cycle at any node or switch in the network. But to reduce complexity a packet is divided into smaller pieces, each of which is eventually transfered in one clock cycle.

Interconnection networks:

- (simple) **bus:** set of wires that provide a single shared path for information transfer. Often used in **UMA** multiprocessors. **Arbitration** is necessary to ensure that only one of many possible requesters is granted use of the bus at any time. **A simple bus does not allow a new request to appear on the bus until the response for the current request has been provided**
- **split-transaction bus:** a request and its corresponding response are treated as separate events and **other transfers may take place between them**. This is usually handled by **associating a unique tag with each request** that appears on the bus. Each response then appears with the appropriate tag so that the source can match it to its original request.
- **ring:** A ring network is formed with point-to-point connections between nodes. A long single ring results in high average latency for communication between any two nodes.
- **bidirectional ring:** halves the latency and doubles the bandiwdth by adding a second ring in the opposite direction, at the expense of more complex communications.
- **hirearchy of rings:** The average latency is reduced without traversing the entire rings, just a section.

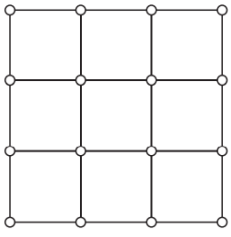


Ring-based interconnection networks.



Crossbar interconnection network.

- **crossbar:** network that provides a direct link between any pair of units connected to the network. It is typically used in UMA multiprocessors to connect processors to memory modules. For n processors and k memories, n × k switches are needed.
- **mesh:**



Nodes in the boundaries and corners have fewer connections,
torus: mesh with wraparound connections between nodes at opposite boundaries of the mesh. So all nodes in a torus have 4 connections (average latency is reduced at the expense of more complexity).

- **snoopy cache:** use of **directories** in each memory module to indicate which nodes may hace copies of a given block in the shared state. Small multiprocessors, including current multicore chips, typically use snooping.
- **mesagge-passing multicomputers:** implementing each node in the system as a complete computer with its own memory. Data that need to be shared are exchanged by sending **messages** from one computer to another.

PARALLEL PROGRAMMING FOR MULTIPROCESSORS

- The compiler cannot automatically identify independent high-level (programming) tasks that could be executed in parallel, it has its **limitations detecting and exploiting parallelism**.
- It is therefore the responsibility of the programmer to explicitly partition the overall computation in the source program into tasks and to specify how they are to be executed on multiple processors.
- **create_thread:** routine library that supports parallel programming. An operating system service is invoked by the library routine to create a new thread with a distinct stack, so that it may call other subroutines and have its own local variables. All global variables are shared among all threads.
- **get_my_thread_id:** library routine that returns a unique integer between 0 and p-1 for each thread. A thread can determine the appropriate subset of the overall computation for which it is responsible
- **barrier:** thread synchronization method that forces a thread to enter into a busy-wait loop until all threads have reached a specific point in the program. This ensures that the threads have completed their respective computations preceding the barrier call.

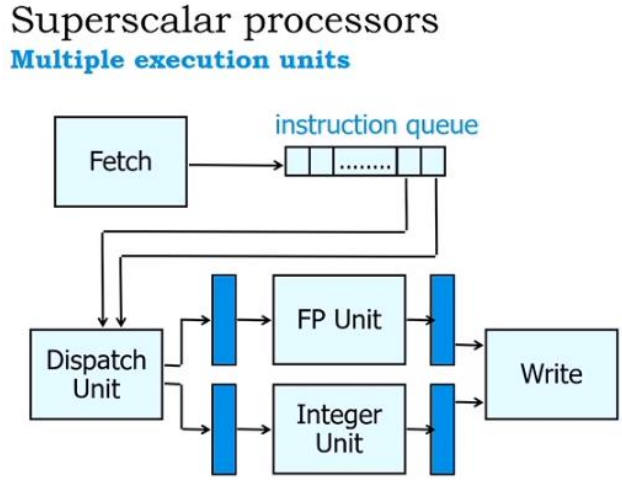
PERFORMANCE MODELING

- The most important measure of the performance of a computer is how quickly it can execute programs
- **execution time:**
  - T_{orig} = current execution time
  - f_{enh} = fraction of execution time affected by enhancent
  - $f_{unenh} = 1 - f_{enh}$ compliment (fraction of execution time not affected by enhancement)
  - $p = f_{enh} * T_{orig}$ = portion of time reduced thanks to enhancement

$$T_{new} = T_{orig} * (f_{unenh} + f_{enh}/p)$$

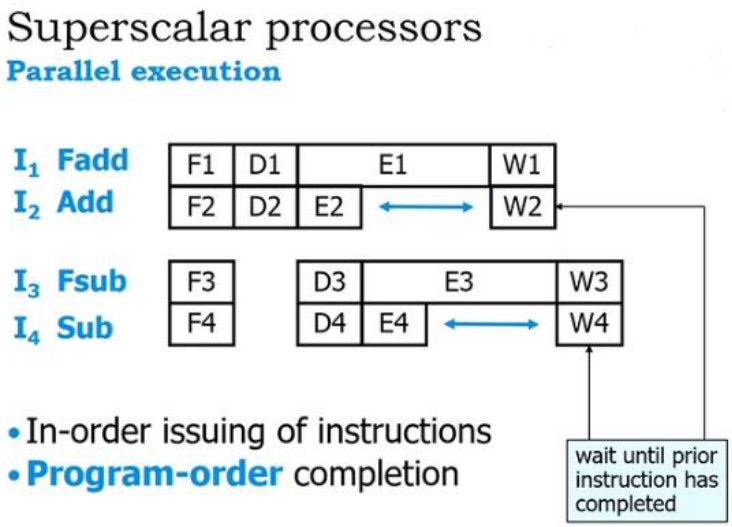
- **speedup = T_{orig}/T_{new} = Amdahl's Law** $= 1/(f_{unenh} + f_{enh}/p)$
  - the benefit of a given performance enhancement increases if it affects a larger portion of the execution time
- **upper bound on the possible speedup:** $1/_{unenh}$
  - $p \rightarrow \infty$ reduction of the fraction fenh of execution time to zero
  - the unenhanced portion of the original execution time can significantly limit the achievable speedup, even if the enhanced portion is improved by an arbitrarily large factor

WEEK 9 - PARALLEL & VIRTUAL MEMORY LECTURE



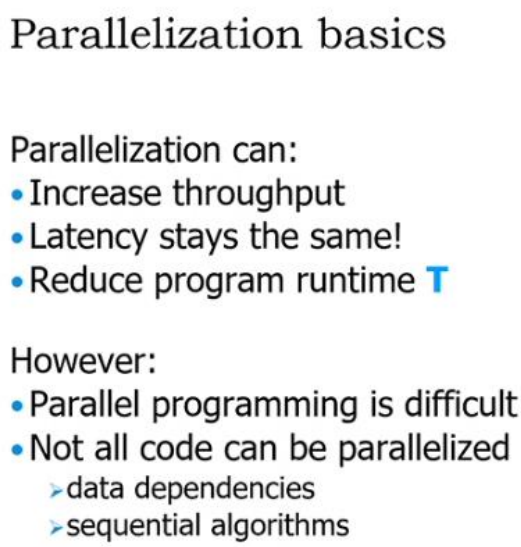
- Superscalar execution: multiple components doing multiple things on paralle
- Floating point unit bottleneck is removed but fetching instructions becomes harder
- instruction que is a bunch of instructions which each can be executed on parallel

.



- In-order issued instructions may be completed at different clock cycles. Therefore there is an extra variant that we add called "program order completion"
- Which will make instructions wait so that the first in first out instruction fetching order is mantained

Amdahl's Law



Data dependencies, some algorithms cant be run on parallel either (min, max, median, which need to check all data)
Only a fraction of a program can be parallelized

Amdahl's law

Time to execute a program sequentially:

$$T_s$$

Time to execute a program using p processors:

$$T_p = T_s \times \left(f_s + \frac{f_p}{p}\right)$$

speedup = T_{orig}/T_{new} = Amdahl's Law $= 1/(f_{unenh} + f_{enh}/p)$

"It takes 10 second to execute sequentially, if we can parallelize 80% of a program and run it on 4 processors what would be our speedup?"

f_p = parallel % = .8

f_s = sequential % = 1-.8 = .2

$T_{orig} = Ts = 10$

$T_{new} = Tp = 10*(.2+.8/4) = 4$

Amdals law = $T_{orig} / T_{new} = 10/4 = 2.5$

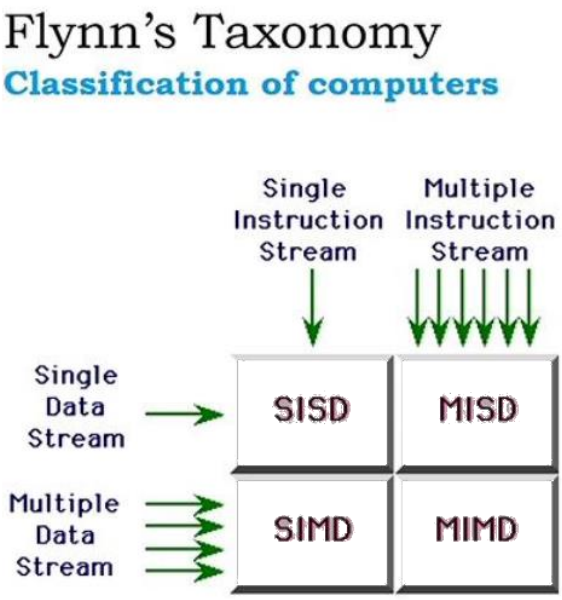
Or just speed up = $1/(f_{unenh} + f_{enh}/p) = 1/(.2+.8/4) = 1/.4 = 2.5$

max speedup = $1/f_s$

$$\lim_{p\to\infty} 1/\left(f_s + \frac{f_p}{p}\right) = 1/(f_s + 0) = 1/f_s$$

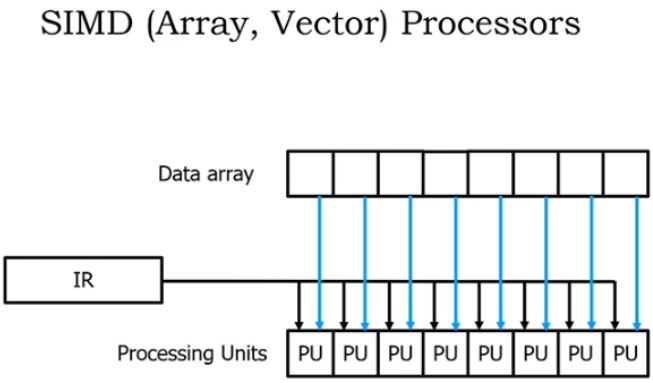
Flynns taxonomy:

- Single Instruction, Single Data (SISD)
  - Conventional system
- Single Instruction, Multiple Data (SIMD)
  - one instruction on multiple data streams
- Multiple Instruction, Multiple Data (MIMD)
  - Multiple instruction streams on multiple data streams
- Multiple Instrucion Single Data (MISD)
  - Multiple instruction streams on single data stream

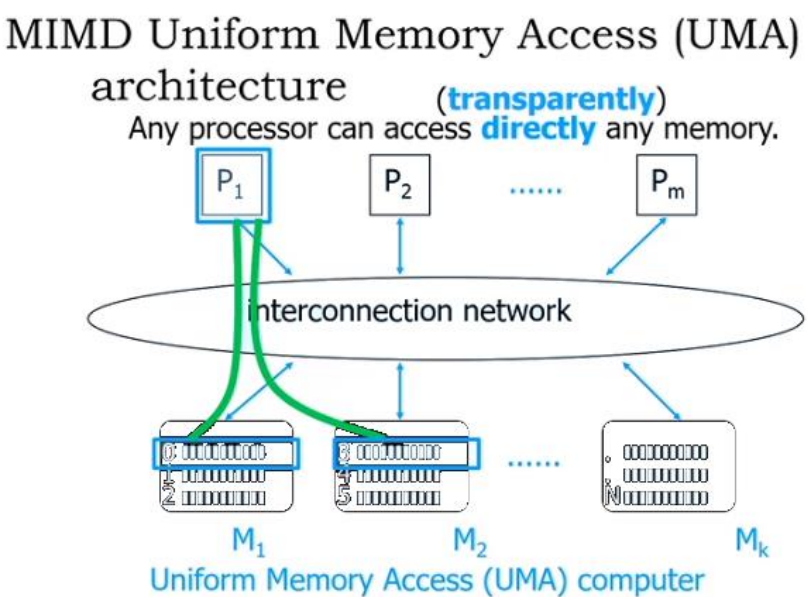


- We can work on parallel with data
- We can have multiple computers (execution units)

SISD: Simple 1 core machine



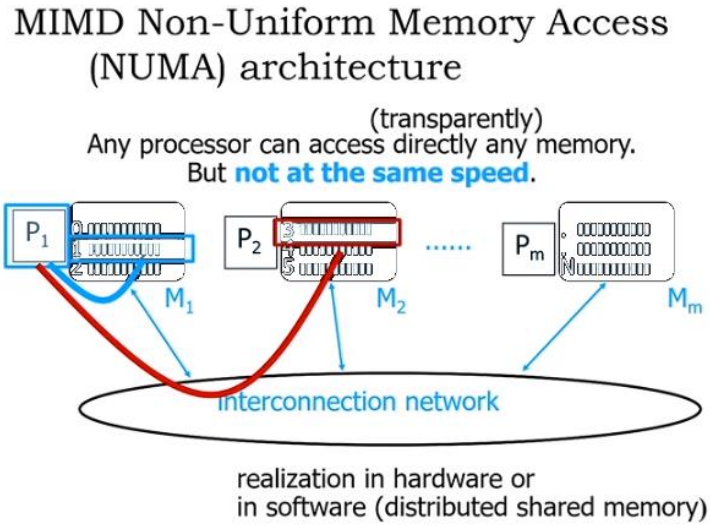
UNIFORM MEMORY ACCESS



We have memory and processors. And whenever we want to grab something from memory the processors go through the interconnection network (bus, ring, mesh etc)

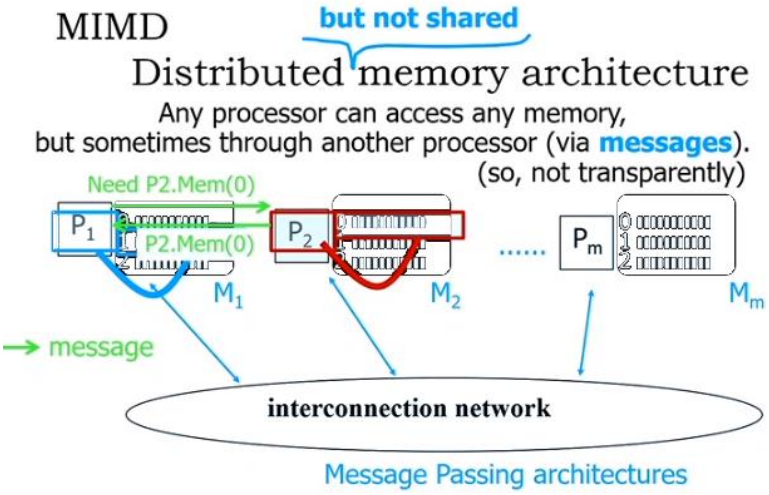
Problem arise when 2 processors want to access the same memory. Therefore we are sequentialising the access to the memory. Eery cache miss goes over the same bus, not efficient.

So here everything can borrow from the common pool but only one at a time.



I can read my own memory very fast, if I want to read someone elses (processor), I can but it goes slow.

It should be up to the program to optimize which memory belongs to which CPU.



Here processor can request memory from another one, but it is not available by default unless the other accepts the request. It's like the internet

Interconnection networks (I/O between processors)

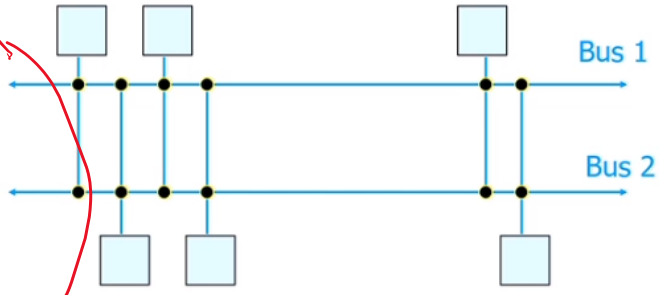
Difficulty in building systems with many processors: the interconnections

Important parameters:

- Diameter: Maximal distance between any two processors
- Degree: Maximal number of connections per processor
- Total number of connections (Cost)
- Bisection width: Largest number of simultaneous messages

Message Passing architectures

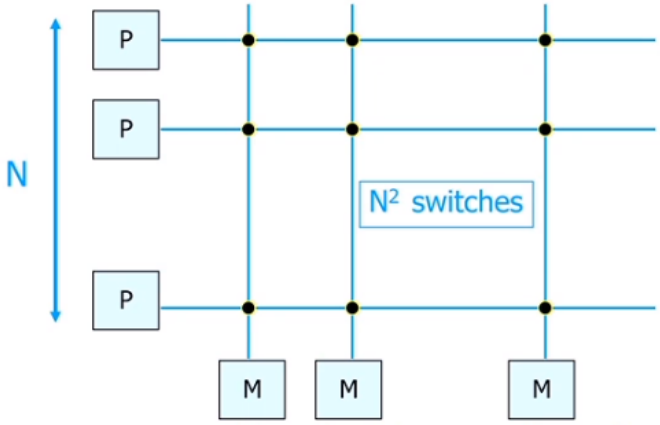
Multiple bus



(Multiple) bus structures

Cross bar

switch



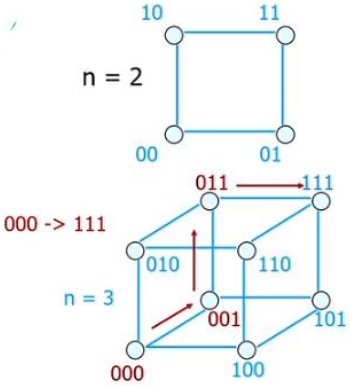
Cross-bar interconnection network

N parallel paths between processors and memory
Simple but still lots of wires

Only works for low scale parallel processing

Hypercubes (1/3)

Non-uniform delay, so for **NUMA** architectures.



$n \times 2^{n-1}$ connections

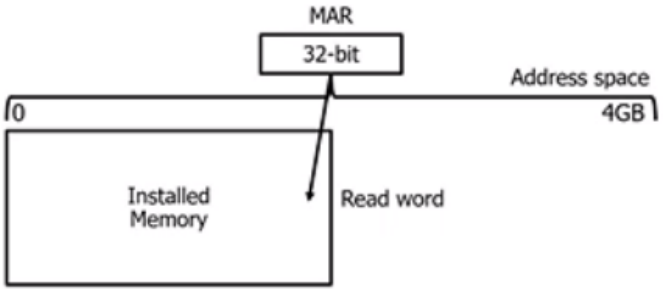
maximum distance n hops

Connected processors differ by 1 bit

Routing:

- scan bits from right to left
- if different, send to neighbor with same bit different
- repeat until end

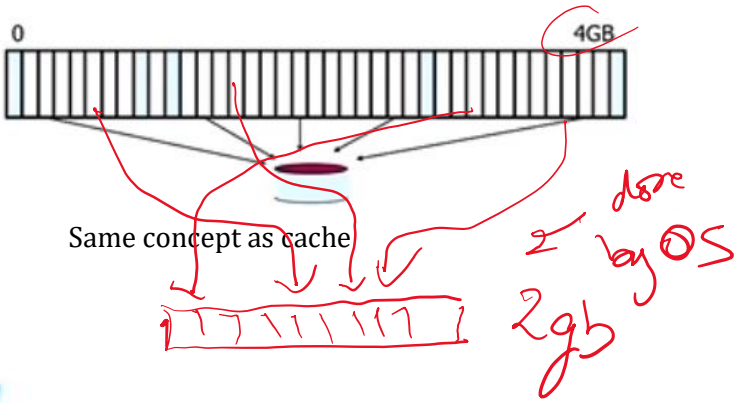
Virtual memory
Why?



It makes handling large files easier.

Virtual memory
Pretend that there is more memory

- Divide address space in chunks
- Store most chunks on disk
- When chunk is needed, move to main memory



Same concept as cache

Virtual memory
Similarity with caching

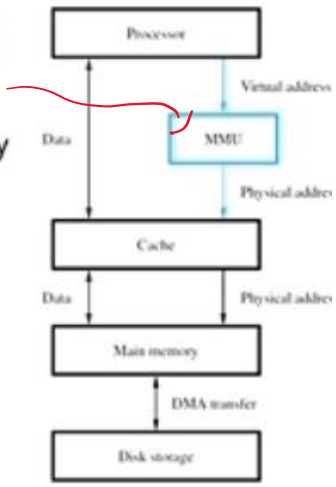
Caching	Virtual Memory
Blocks	Pages
Cache miss	Page fault
Tag field	Page number
Byte/word field	Page offset
Mapping function	Page table

We want to give the programmer the illusion that he is always to write on all those 4GB although we only have 2GB available.

We would need to map the 4gb pages into the 2GB page table

Virtual memory
Hardware/software support

- Memory Management Unit
  - mapping of pages in memory
  - access control
- Operating System
  - set up translation
  - handle misses



Processor

Bus Virtual address

MMU

Bus Physical address

Cache

Physical address

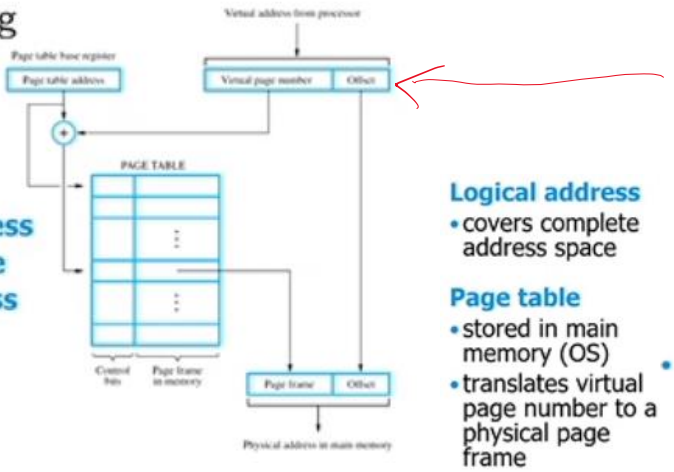
Main Memory

DMA controller

Disk storage

Caches do everything with hardware, Virtual memory also has software support. It gives us more flexibility

Demand paging



- From **virtual address**
- through **page table**
- to **physical address**

Logical address
- covers complete address space

Page table
- stored in main memory (OS)
- translates virtual page number to a physical page frame

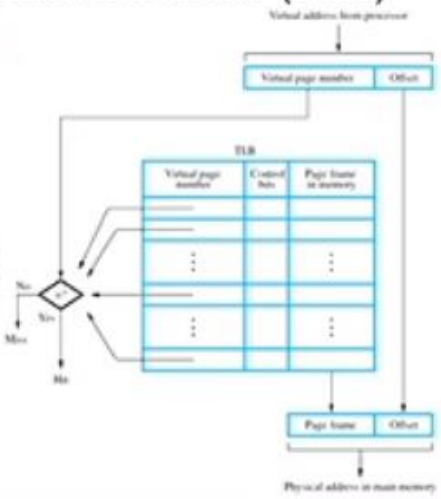
- CPU contains special register that stores the starting location of the page table

1. We start with a logical/virtual address where the program thinks he can store anything
2. Page table provides us with a mapping in the page addres table register
3. Because you need to have this table available for each program and you dont want them to overwrite memories across so one page table is created per running program.
4. There is a control bit for writing confirmation.
    a. Generally we will not copy things to the disk every time, just before closing the program.

Translation Lookaside Buffer (TLB)

Locality of reference

- Special cache for small part of the page table
- Look up virtual page number to find page frame



TLB stores the recent translations of virtual memory to physical memory like a cache.

So instead of having to look up the whole page table in main memory every time, you have the TLB quickly by hand.

404 Page **Frame** Not Found

- If page frame not found in TLB
  - regular cache miss
- If page frame not found in page table
  - page fault!

Where is the page? The page is on Disk, it was virtual... So it is the job of the OS to bring the page from disk to main memory into the page table because we have now a new mapping.
and Disk stuff is moved via the DMA (and its bus).

Handling page faults

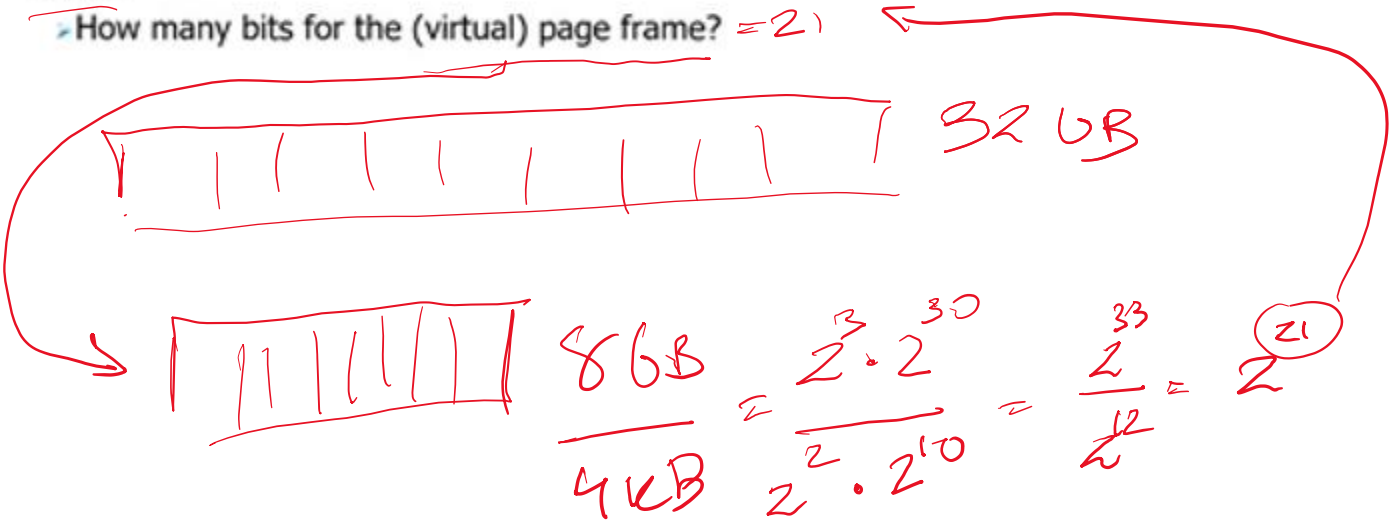
- MMU issues a trap (software interrupt)
- CPU
  - aborts the instruction
  - switches to kernel mode
- OS
  - searches for a free page frame
  - loads the requested page from disk into memory (context switch)
  - updates the page table
  - restarts the aborted instruction

- Caches sometimes use write-through.
  - Would you use the same technique for the page table to disk? Why?

NO! We want to minimize the number of times we talk to the disk because the disk is extremly slow

8 GB of word addressable RAM, and an OS that supports memory up to 32 GB and uses pages of 4 KB.

- How many bits for the (virtual) page frame? = 21



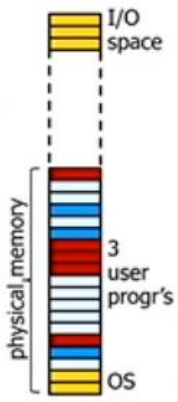
Page table for each process

Operating System loads correct page table for each process

- OS maps only what is allowed
- Other parts of the memory become unreachable!

Separate 'kernel' mode for OS processes

- Gives access to special instructions:
  - access 'page-table base register', ...



PTBR = page table for each process ensures that the allocated pages for a program are fixed and a program cant mess with the pages of another program, such us the OS, which is very protected. Furthermore, only the OS should be allowed to write the page table base register.