Jin Wei CS2100 Notes

Powers of 2

10	9	8	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-6
1024	512	256	128	64	32	16	8	4	2	1	0.5	0.25	0.125	0.0625	0.3125	0.015625

von Neumann Architecture
- stores program & data in memory

$$\left.\begin{array}{l} b \wedge b = 0 \\ b \wedge 0 = b \\ a \wedge (b \wedge b) = a \wedge 0 = a \end{array}\right\} \text{for swapping int}$$

C programming

$$\left.\begin{array}{l} \text{int : 4 bytes} \\ \text{float : 4 bytes} \\ \text{double : 4 bytes} \end{array}\right\} \text{depends on CPU.}$$

char : always 1 byte

(int) float
→ truncates

8bits = 1 byte

32 → $(32)_{10}$
032 → $(32)_8$ ← MIPS no octal input
0x32 → $(32)_{16}$

Variable has:
- name
- data type
- value
- address

Everything in C is pass-by-value <u>not</u> reference.

```
                    ↙ address
scanf ("string", x1, x2, ...)
printf ("string")
printf ("string", x1, x2, ...)
puts (str)  // w/ newline
```

%c char

%d int

%f float (double uses %f for output)

%lf double (for input)

%p address ← different execution, different address

%zu size-t

Base-R to decimal

$$\underset{3\ 2\ 1\ 0\ \ -1\ -2\ -3}{1101.101_2} = 1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0 + 1\times2^{-1} + 0\times2^{-2} + 1\times2^{-3}$$

Decimal to Base-R

Whole numbers : repeated division by R
→ division ∵ $R^k \to R^{k-1}$

```
2 | 43
2 | 21  rem 1  ← LSB
2 | 10  rem 1
2 | 5   rem 0
2 | 2   rem 1
2 | 1   rem 0
  | 0   rem 1  ← MSB
    ↖ repeat til 0
```

43 = $(101011)_2$

Fractional portion : repeated multiplication
→ multiplication ∵ $R^{1-k} \leftarrow R^{-k}$

```
                  take this
                     ↓
0.3125  × 2 =  0.625    ← MSB
0.625   × 2 =  1.25
0.25    × 2 =  0.5
0.5     × 2 =  1        ← LSB
               ⏟
           repeat til 0
       ( might not terminate
         but will have pattern )
```

0.3125 = $(0.0101)_2$

R1 → R2 ≡ R1 → decimal → R2

but if $R1^n = R2$
→ partition into groups of n

$$(\underset{2}{10}\ \underset{7}{111}\ \underset{3}{011}\ \underset{1}{001}\ .\ \underset{5}{101}\ \underset{6}{110})_2 = (2731.56)_8$$

n ← . → n

Sign-extension
↳ pad with MSB

Zero-extension
↳ pad with 0s

max int is $2^{31}-1$
= 2 147 483 647
for 2's complement

Fixed-Point Representation

- fixed number of bits for integer & fractional components

Sign and Magnitude

□ magnitude
sign

0 → +ve
1 → -ve

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

negation: flip sign bit

$$A - B = A + (-B)$$

1's complement

: diminished radix complement

$-X = 2^n - X - 1$

$X = r^n - r^{-m} - X$
fractional

MSB is sign bit

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

negation: flip <u>all</u> bits

e.g. FFFF - 0001 = FFFE

→

1. Binary addition on $a + b$
2. If MSB carry out, add 1 to result
3. Check overflow.

Overflow if MSB 1+1 = 10

$$\begin{array}{rc|c|ccc} & & 1 & 1 & 0 & 1 \\ + & & 1 & 0 & 1 & 0 \\ \hline & 1 & 0 & 1 & 1 & 1 \end{array}$$

2's complement

: radix complement

e.g. 10000 - 0001 = FFFF

$-X = 2^n - X$

MSB is sign bit

no duplicate 0

range: $[-2^{n-1}, 2^{n-1}-1]$

negation: flip <u>all</u> bits, +1 to LSB (incl. fractional)

→

1. Binary addition on $a + b$
2. Ignore MSB carry out
3. Check overflow.

Floats IEEE 754

normalised w/ implicit leading bit & pad w/ 0s to the right

| sign | exponent | mantissa |

defn: fractional component

all 0s is a "denormal form" that represents 0.
→ cannot be represented precisely otherwise

single-precision (32-bits)
- 1-bit
- 8-bit → excess-127
- 23-bit

double-precision
- 1-bit
- 11-bit → excess-1023
- 52-bit

1. Convert |f| to binary
   → repeated mul / div
2. Normalise to $\pm 1.\text{mantissa} \times 2^{\text{exponent}}$
3. Sign: 0/1 for +ve / -ve
4. Merge

Multiply:
1. Multiply mantissa
2. Add exponents
3. Normalise

Addition:
1. Equalise exponents
2. Add new mantissa
3. Normalise

Excess - N

on M bits

start at -N

$(x)_{\text{binary}} + N = (x)_{\text{Excess-N}}$

→ easy to compare values

Excess-4	Value
000	-4
001	-3
010	-2
011	-1
100	0
101	1
110	2
111	3

Pointers

→ ++ and -- based on sizeof(type)

```
int a = 123;
&a → addr of a
int *a_ptr = &a;
    *a_ptr = 123 = a
```

pointers

```
void swap(int *a, int *b)
```

Arrays

: homogeneous collection of data

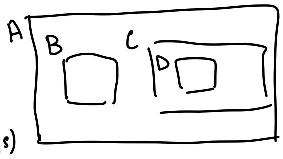
- contiguous in memory

values.length > size ⇒ warning

```
int x[4] = {1}; → x[0] = 1
                  x[1] = 0 ⎫
                     ⋮     ⎬ defaut value
x = &x[0]         x[3] = 0 ⎭
arr[i] = *(arr + i)
arr2 = arr1; X CANNOT do assignment
```

Structures

- can be nested
- allows heterogeneous members
- allows assignment (unlike arrays)



```
typedef struct {
    int a,b;
    float f;
} box_t;

box_t box = {0,1,0.3};
box.f; → 0.3
```

dot

```
void func(box_t box){...}
func(&box);
```

use address to use "reference".

```
(*box_ptr).b ≡ box_ptr → b
```

arrow

Strings

size = string length + 1 ← '/0'

ends with '/0' → ascii of 0.

```
char str[] = "abc"; ≡ char str[] = {'a','b','c','/0'}
```

stdin

```
fgets(str, size, file)
```

reads up to size OR newline

↳ | e | a | t | \n | \0 |

should change to \0

```
scanf("%s", str)
```

- until whitespace

<string.h>

1. strlen(s)
   → no. of char
2. strcmp(s1, s2)
   → s1 lexicographical comparison
3. strncmp(s1, s2, n)
   → up to \0 or n
4. strcpy(s1, s2)
5. strncpy(s1, s2, n)

MIPS

add \$s0, \$t1, \$t2 ≡ \$s0 = \$t1 + \$t2

immediate : $[-2^{15}, 2^{15}-1]$ ← 16-bit 2's complement

↖ zero-extension

for large constants (> 16 bits)

lui \$t0, 0xAAAA

ori \$t0, \$t0, 0xBBBB # \$t0 = 0xAAAABBBB

lw \$t0, 4(\$s0)

dest ↗ ↑ offset (multiple of 4, can be 0) ↑ address

sw, lb, sb → offset don't need to be multiple of 4.

\$t0 = mem[\$s0 + 4]

beq } \$r1, \$r2, Label ← relative
bne }

j Label ← absolute

slt \$t0, \$s0, \$s1 if \$s0 < \$s1,
\$t0 = 1
else
\$t0 = 0

Amdahl's Law
→ make frequently used instructions FAST!

Instruction Set Architecture (ISA)

① Complex instruction set computer (CISC) ← x86_32 (IA32)

② Reduced instruction set computer (RISC)
↑ MIPS, ARM

Storage Architectures

① Stack

② Accumulator
- one operand implicitly on accumulator

③ General purpose
- register-memory
- register-register (MIPS)

④ Memory-memory
- go through BUS (slow)
- small instruction set

Instruction Length

Variable vs Fixed vs Hybrid (≈ variable)
→ OPCODE (must have!)
→ operands
→ usually 2^n sizes

Work w/ most constrained instruction type.
↑ most operands

R-format

			dest	shift	
OPCODE = 0	rs	rt	rd	shamt	funct
6	5	5	5	5	6

add \$rd, \$rs, \$rt

sll/srl \$rd, \$rt, shamt ← \$rs = 0

I-format

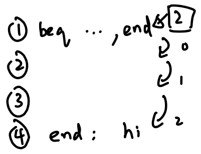
sign-extended except andi, ori, xori

		dest	
OPCODE	rs	rt	IMMEDIATE
6	5	5	16

beq/bne \$rs, \$rt, immd → up to ± 2^{17} bytes

addi \$rt, \$rs, immd if jump: PC = PC + 4 + (immd × 4)

lw \$rt, immd(\$rs)


① beq ..., end ← $\boxed{2}$
②
③
④ end : hi

J-format

OPCODE	TARGET
6	26

↑ $(2)_{10} = (000010)_2$

256MB boundary

$2^{26} - 1$ - (instructions between)

4 MSB of next PC	+ target	+ 00

if at 0xAFFFFFFE boundary, cannot use j.

Endians

Big Endian

→ MSB in lowest address

0x DE AD BE EF

0	DE
1	AD
2	BE
3	EF

Little Endian

→ LSB in lowest address

0x DE AD BE EF

0	EF
1	BE
2	AD
3	DE

for $a_1 < a_2 < \cdots < a_n$,

$$\text{min}: 2^{a_1} + \sum_{k=1}^{n-1} 2^{(a_{k+1} - a_k)} - n + 1$$

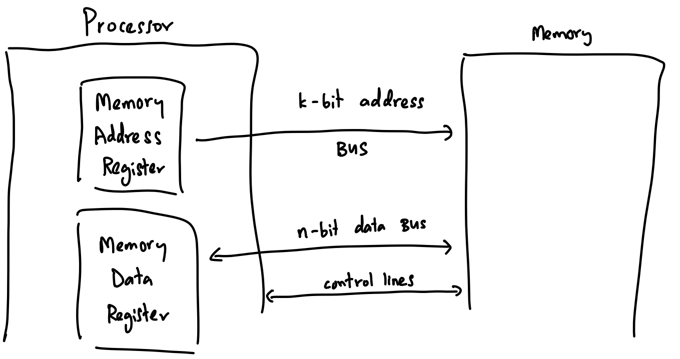
$$\text{max}: 2^{a_n} - \sum_{k=1}^{n-1} 2^{(a_n - a_k)} + n - 1$$

largest opcode ↙

$$\text{Max}: 2^{M} \underbrace{- 2^{A} + 1}_{\text{repeat}}$$

	6	5	5
A	OP	FUNCT	ADDR
	6	5	5
B	OP	ADDR	ADDR

$2^{11} - 2^5 + 1$
$= 2017$ //



* up to 2^k addressable locations

Control & Datapath

1. Fetch
   - get instruction from memory
   - get address from PC register
2. Decode
   - find which operation
3. Operand fetch
   - get operands needed for operation
4. Execute
5. Writeback
   - store results in register

PC read during first half of clock period
& updated at rising clock edge

	RegDst	ALUSrc	MemToReg	RegWrite	MemRead	MemWrite	Branch	ALUOp
R-type	1	0	0	1	0	0	0	10
lw	0	1	1	1	1	0	0	00
sw	x	1	x	0	0	1	0	00
beq	x	0	x	0	0	0	1	01

RegDst

0 : Write register = inst [20 : 16] ← I - type
1 : Write register = inst [15 : 11] ← R - type

RegWrite

0 : no register write

1 : write to register

ALUSrc

0 : Operand 2 = RD2
1 : Operand 2 = SignExt (inst [15:0]) ← lw/sw

MemRead

0 : no memory read
1 : memory read using Address (ALU result)

MemWrite

0 : no memory write
1 : mem [Address] ← write data (RD2)

MemToReg

☆ 1 : WD = memory read data
0 : WD = ALU result

PCSrc / Branch

← PCSrc = Branch + isZero

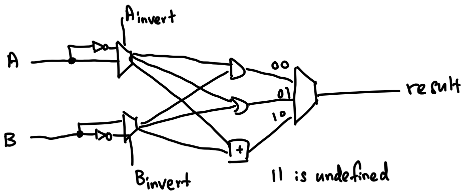
0 : next PC = PC + 4

1 & isZero : PC + 4 + SignExt (inst [15:0]) << 2

ALUControl

0000	AND
0001	OR
0010	ADD
0110	SUB → A − B = A + (−B) = A + ~B + 1 ← Cin
0111	SLT
1100	NOR

Ainvert Binvert



11 is undefined

Boolean Algebra

Identity

$A + 0 = 0 + A = A$ $\quad A \cdot 1 = 1 \cdot A = A$

Inverse / Complement

$A + A' = A' + A = 1$ $\quad A \cdot A' = A' \cdot A = 0$

Commutative

$A + B = B + A$ $\quad A . B = B \cdot A$

Associative

$A + (B + C) = (A + B) + C$ $\quad A \cdot (B \cdot C) = (A \cdot B) \cdot C$

Distributive

$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$ $\quad A + (B \cdot C) = (A + B) \cdot (A + C)$

Idempotency

$X + X = X$ $\quad X \cdot X = X$

One element / Zero element

$X + 1 = 1 + X = 1$ $\quad X \cdot 0 = 0 \cdot X = 0$

Involution

$(X')' = X$

Absorption 1

$X + X \cdot Y = X$ $\quad X (X + Y) = X$

Absorption 2

$X + X' \cdot Y = X + Y$ $\quad X \cdot (X' + Y) = X \cdot Y$

De Morgan → generalisable to >2 variables

$(X + Y)' = X' \cdot Y'$ $\quad (X \cdot Y)' = X' + Y'$

Consensus

$X \cdot Y + X' \cdot Z + \underline{Y \cdot Z} = X \cdot Y + X' \cdot Z$ $\quad (X + Y) \cdot (X' + Z) \cdot \underline{(Y + Z)} = (X + Y) \cdot (X' + Z)$

Duality

$x + 1 = 1 \iff x \cdot 0 = 0$

inverts

$$\boxed{\begin{array}{l} + \text{ and } \cdot \\ 0 \text{ and } 1 \end{array}}$$

Minterm of n variables : $\overbrace{x_1 \cdot x_2 \cdot \cdots \cdot x_n}^{\text{must have all n literals}}$

Maxterm of n variables : $x_1 + x_2 + \cdots + x_n$

up to 2^n possibilities

$mx = (Mx)'$

x	y	Minterms	Maxterms
0	0	$x' \cdot y'$ m0	$x + y$ M0
0	1	$x' \cdot y$ m1	$x + y'$ M1
1	0	$x \cdot y'$ m2	$x' + y$ M2
1	1	$x \cdot y$ m3	$x' + y'$ M3

$\overbrace{\text{sum of minterms}}^{\text{don't compress}}$ = canonical sum of products (SOP)

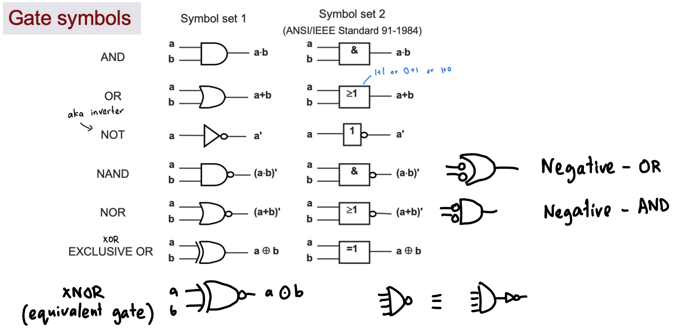
product of maxterms = canonical product of sums (POS)

$F = m1 + m2 = \Sigma m(1,2) = \Sigma m(1-2)$

$G = M0 + M4 = \Pi M(0,4)$

Σm and ΠM can convert with De Morgan's

Logic Circuits



A	B	AND	OR	XOR	NAND	NOR	XNOR
0	0	0	0	0	1	1	1
0	1	0	1	1	1	0	0
1	0	0	1	1	1	0	0
1	1	1	1	0	0	0	1

fan - in : number of inputs of a gate

* assume complemented literals don't exist unless stated for CS2100

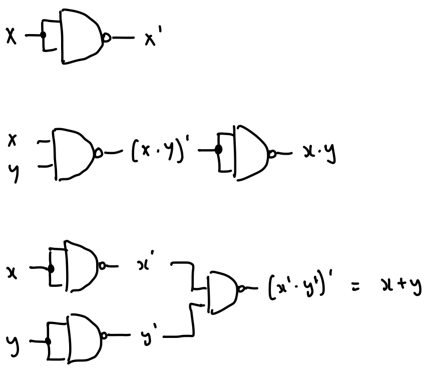
Universal Gates

Complete sets of logic : {AND, OR, NOT}, {NAND}, {NOR}, {AND, NOT}, {OR, NOT}, ... (a lot)

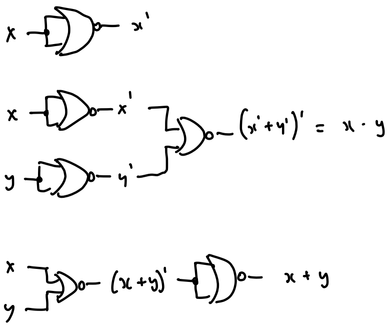
Others still useful

- XOR : parity bit generation
- economical (save on gates)

NAND



NOR



SOP can easily be implemented with :

1) 2-level AND-OR circuit
2) 2-level NAND circuit

↑ invertor not considered

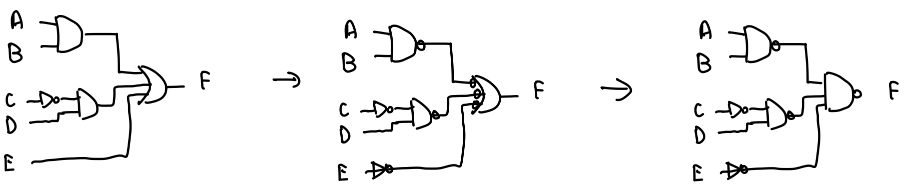
POS can easily be implemented with :

1) 2-level OR-AND circuit
2) 2-level NOR circuit

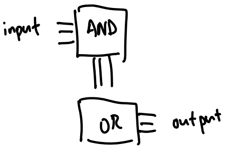
↑ invertor not considered

e.g. F = A·B + C'·D + E

AND-OR to NAND



Programming Language Array (PLA)



- may not be able to implement every mapping

Read Only Memory (ROM)

- Similar to PLA
- fully decoded : able to implement any mappings

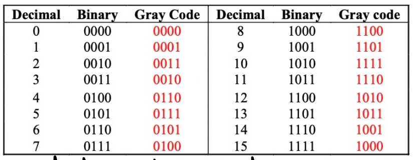
Simplification

- minimise literals
- minimise terms used

Gray Code

aka reflected binary code

- unweighted (opposite of arithmetic code)
- only 1-bit change (max $\xrightarrow[\text{back}]{\text{loops}}$ 0)
- good for error detection
- n bits → 2^n values

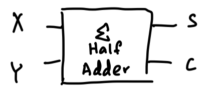


Decimal	Binary	Gray Code	Decimal	Binary	Gray code
0	0000	0000	8	1000	1100
1	0001	0001	9	1001	1101
2	0010	0011	10	1010	1111
3	0011	0010	11	1011	1110
4	0100	0110	12	1100	1010
5	0101	0111	13	1101	1011
6	0110	0101	14	1110	1001
7	0111	0100	15	1111	1000

standard 4-bit gray code

many possible implementations

Half-Adder



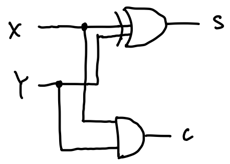
(X + Y)

X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$C = X \cdot Y = m3$$

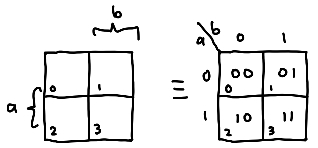
$$S = X' \cdot Y + X \cdot Y = m1 + m2$$

$$= X \oplus Y$$



Karnaugh - Maps

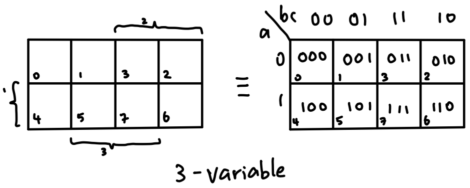
- each square represents a minterm
- 2 adjacent squares ⇒ differ by one literal
- wrap around exists
- n - variable ⇒ n neighbours
- 2^n cells
- for 2^x adjacent minterms, n - x literals



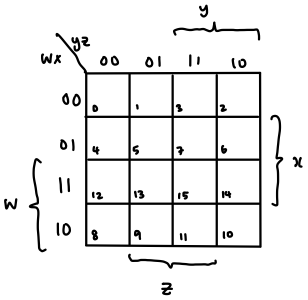
0 = b'

1 = b

2 - variable

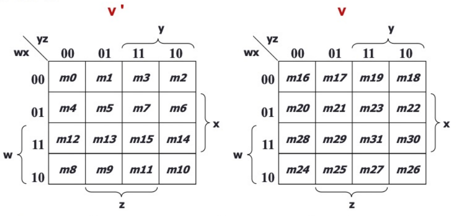


3 - variable



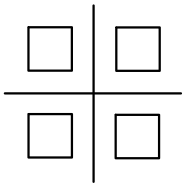
4-variable

- Organised as two 4-variable K-maps. One for v' and the other for v.



Corresponding squares of each map are adjacent.
Can visualise this as *one 4-variable K-map* being on TOP *of the other 4-variable K-map*.

5 - variable



6 - variable

Groupings in K-Maps

based on A + A' = 1 (Unifying Theorem)

1. group as many as possible
   → PI
   → results in less terms
2. ignore redundant groups
   → for EPI

* remember wrap arounds

implicant: product term that can be used to cover minterms of the function

EPI: PI that includes ≥ 1 minterm not covered by any other PI.

→ does NOT need to cover all 1s

Don't Care

- denote with X or d
- use Σd or ΠD

→ include in PI/EPI only if the PI/EPI contains 1.

finding SOP

→ all EPIs + remaining PIs not covered by any EPIs

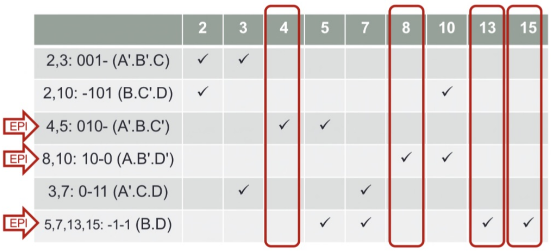
Quine - McCluskey

- similar to K-map but not limited to ~ 6 variables

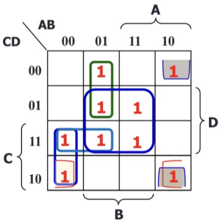
Steps:

1. list out all minterms in groups with same number of 1s
2. Combine codes that differ by 1 bit into bigger group, write combined with "-"
3. Repeat step 2 and continue combining

Identify EPIs (columns containing single tick)



		2	3	4	5	7	8	10	13	15
	2,3: 001- (A'.B'.C)	✓	✓							
	2,10: -101 (B.C'.D)	✓						✓		
EPI	4,5: 010- (A'.B.C')			✓	✓					
EPI	8,10: 10-0 (A.B'.D')						✓	✓		
	3,7: 0-11 (A'.C.D)		✓			✓				
EPI	5,7,13,15: -1-1 (B.D)				✓	✓			✓	✓



CD \ AB	00	01	11	10
00		1		1
01		1	1	
11	1	1	1	
10	1			1

Eliminate EPI rows

Answer = EPIs + remaining ticks

Combinational Circuit

Each output depends entirely on the immediate inputs
→ like mathematical functions
→ no internal state

To analyse:

1. Label inputs/outputs
2. Obtain functions of intermediate points & outputs ↙ after gates/blocks
3. Draw truth table x x
4. Deduce/guess functionality

Gate-level Design

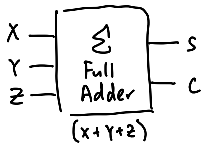
- with logic gates OR, AND, NOT, XOR, ...
- analyse w/ K-map or truth table to get SOP (& simplify) to build circuit

Block-level Design

- with functional blocks
- using simpler blocks to build complex blocks
- find common patterns & intermediate states

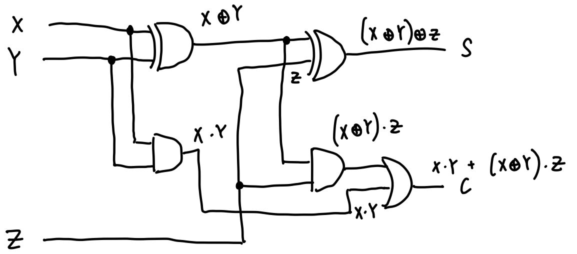
- Reduce cost
- Increase speed
- Design simplicity (re-use if possible)

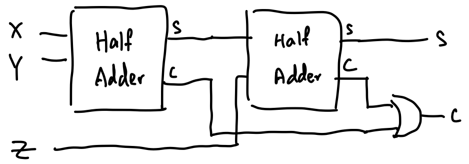
Common Blocks

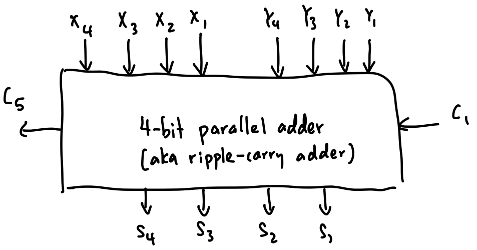


$S = X \oplus (Y \oplus Z)$

$C = X \cdot Y + (X \oplus Y) \cdot Z$

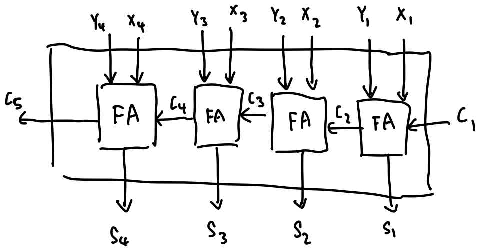






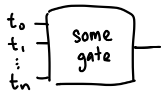
gate-level too complicated to analyse

→ 2^9 possible inputs



→ can build 8, 16, ... bit adders

Circuit Delays



delay of this gate

$\overset{\downarrow}{t} + \max(t_0, t_1, \cdots, t_n)$

→ have to wait for all inputs to be stable

↙ medium-scale integrations

MSI Components

- An integrated chip (IC) is a set of electronics on one small flat piece/chip of semiconductor material.
- Scale of integration: # of components fitted on a standard IC chip

n:m Decoder

or n × m or n-to-m

n: # of input bits

m: # of possible outputs $[m \leq 2^n]$ ← usually, $m = 2^n$

→ input order matters (not always but sometimes)

2:4 decoder

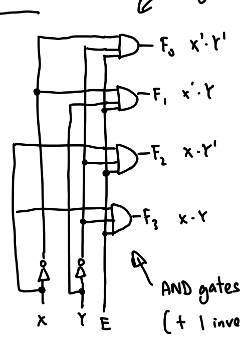
E	X	Y	F_0	F_1	F_2	F_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	d	d	0	0	0	0

↑ 1-enabled

$\underbrace{\quad\quad}_{\text{minterms}}$

∴ can use to implement SOP functions with OR gates

only 1 high/low



AND gates only (+ 1 invertor)

0-enabled usually denoted with $\bar{E}$ or E'

→ more commonly used in MSI decoders

→ truth table just swap 0 & 1

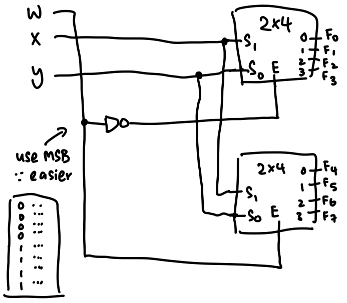
active-high (normal output)

→ 1 if true

active-low (negated output)

→ 0 if true

can build $(n+1):2^{n+1}$ decoder with 2 $n:2^n$ decoders



use MSB ∵ easier

74138 (3:8 decoder)

→ very common

→ G1 = 1, G2A = 0, G2B = 0 to enable

→ active-low output

m:n Encoders

→ converse of decoders

→ only 1 input bit is high

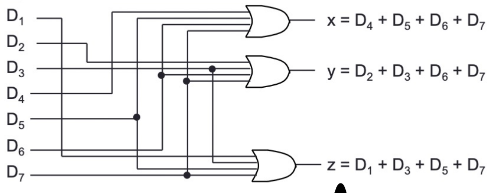
(or in the case of priority encoders, highest priority is chosen)

↑ (all 0 is invalid!)

m : # of input bits $[m \leq 2^n]$

n : # of outputs

8:3 encoder


D_0
D_1
D_2
D_3
D_4
D_5
D_6
D_7
$x = D_4 + D_5 + D_6 + D_7$
$y = D_2 + D_3 + D_6 + D_7$
$z = D_1 + D_3 + D_5 + D_7$

↑ use truth table / binary

$1 : 2^n$

Demultiplexer or demux

- → take input & a set of selection lines
- → directs input to one of the output lines
- → identical circuit as decoder with enable

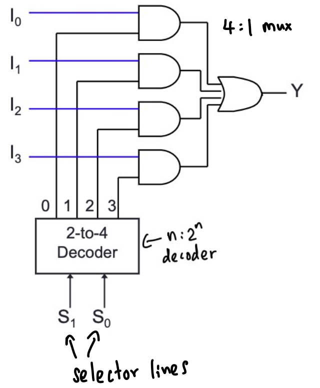
$2^n : 1$

Multiplexer or mux or data selector

- → many inputs & a set of selection lines
- → chooses one of the inputs as output

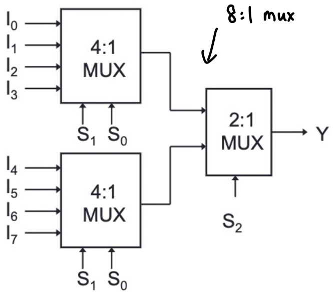
output = $I_0 \cdot m_0 + I_1 \cdot m_1 + \cdots + I_{n-1} \cdot m_{n-1}$

→ use $n : 2^n$ decoder & AND with the input bits (as above)



	A B C D	AB	AC	AD	BC	BD	CD
0	0000	00	00	00	00	00	00
1	0001	00	00	01	00	01	01
2	0010	00	01	00	01	00	10
3	0011	00	01	01	01	01	11
4	0100	01	00	00	10	10	00
5	0101	01	00	01	10	11	01
6	0110	01	01	00	11	10	10
7	0111	01	01	01	11	11	11
8	10 00	10	10	10	00	00	00
9	10 01	10	10	11	00	01	01
10	10 10	10	11	10	01	00	10
11	10 11	10	11	11	01	01	11
12	11 00	11	10	10	10	10	00
13	11 01	11	10	11	10	11	01
14	11 10	11	11	10	11	10	10
15	11 11	11	11	11	11	11	11

Merging mux



74151A 8-to-1 mux

- → 0-enabled (strobe = enable)
- → both active-high & active-low

→ group by common term in truth table

S_2	S_1	S_0	Y
0	0	0	I_0
0	0	1	I_1
0	1	0	I_2
0	1	1	I_3
1	0	0	I_4
1	0	1	I_5
1	1	0	I_6
1	1	1	I_7

to implement SOP function { put 1 if it is a minterm of the function else 0

& variables = select

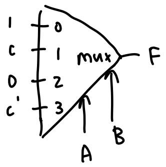
→ BUT we can do BETTER!!

→ use $2^{n-1} : 1$ mux is sufficient

e.g. $F(A,B,C) = \Sigma m(0,1,3,6)$

group by AB ∵ using C ← can use any

A	B	C	F	mux
0	0	0	1	1
0	0	1	1	
0	1	0	0	C
0	1	1	1	
1	0	0	0	0
1	0	1	0	
1	1	0	1	C'
1	1	1	0	



Sequential Circuit

→ has internal state/memory

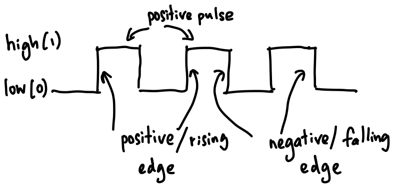
↙ 1-bit data

memory element : device that remembers a value indefinitely , or change value on command from inputs

$Q(t)$ or Q : current state

$Q(t+1)$ or Q^+ : next state

Clock ◷ - square wave



pulse - triggered
- latches

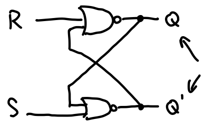
ON = 1 , OFF = 0

edge - triggered
- flip - flops
- positive edge-triggered : ON = rising edge , OFF = other time
- negative edge-triggered : ON = falling edge , OFF = other time

S-R Latch

or R-S

(active-high)



complementary outputs

use NOR gates

need to "loop" circuit until stable (≤ 3 loops)

Characteristic Table

S	R	Q	Q'	
0	0	NC	NC	
0	1	0	1	Latch SET
1	0	1	0	Latch RESET
1	1	0	0	Invalid input

S	R	Q^+	
0	0	Q	
0	1	0	
1	0	1	
1	1	N/A	← breaks circuit

→ $Q^+ = S + R' \cdot Q$

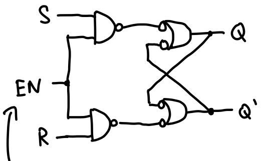
active-low variant

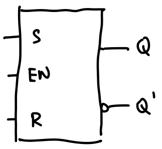
- use NAND or negative-or instead of NOR
- sometimes, S and R are labelled S' and R'

S	R	Q^+
0	0	N/A
0	1	1
1	0	0
1	1	Q

Gated S-R Latch

← gated = enable (EN)

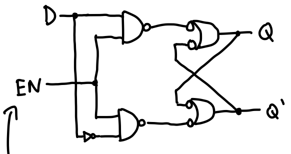


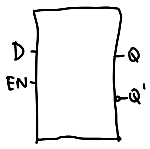


1-enabled

Gated D Latch

aka data latch





1-enabled

→ avoids having invalid input

EN	D	Q^+	
1	0	0	Reset
1	1	1	Set
0	x	Q	No change

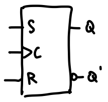
when EN = 1, $Q^+ = D$

↑ if non-gated, D latch pretty useless

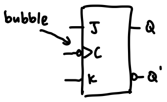
Flip Flops

← use this over latch (only difference is the source of enable)

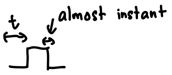
- synchronous (by clock)
- bistable
- output changes state at a specified point based on a clock
    - rising edge
      OR
    - falling edge



positive edge-triggered
S-R flip-flop



negative edge-triggered
J-K flip-flop



* use edge ∵ period of time is very short compared to period of high/low

* choosing positive or negative edge-triggered is arbitrary

* flip-flops below are positive edge-triggered

S-R flip-flop

X - don't care

↑ - rising edge

S	R	Clock	Q^+	
0	0	x	Q	no change
0	1	↑	0	reset
1	0	↑	1	set
1	1	↑	N/A	invalid

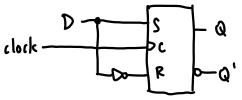
→ $Q^+ = S + R' \cdot Q$

D flip-flop

D	Clock	Q^+	
1	↑	1	set
0	↑	0	reset

→ $Q^+ = D$

→ used for parallel data transfer

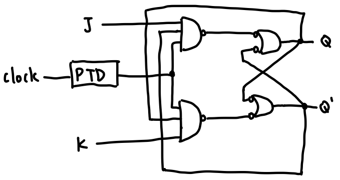


build D flip-flop using S-R flip-flop

J-K flip-flop

J	K	Clock	Q^+	
0	0	X/↑	Q	no change
0	1	↑	0	reset
1	0	↑	1	set
1	1	↑	Q'	toggle

→ $Q^+ = J \cdot Q' + K' \cdot Q$



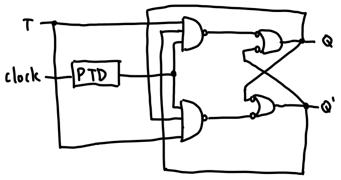
PTD = Pulse transition detector

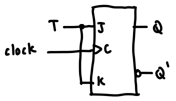
T flip-flop

T	Clock	Q^+	
0	↑	Q	no change
1	↑	Q'	toggle

→ $Q^+ = T' \cdot Q + T \cdot Q'$

$= T \oplus Q$





build T flip-flop using J-K flip-flop

Asynchronous Inputs

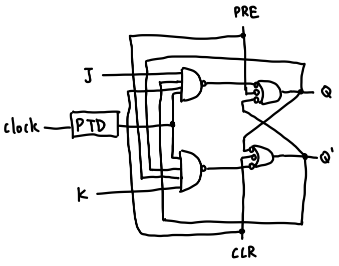
PRE (preset) = active, Q is immediately set to 1 } ignores clock
CLR (clear) = active, Q is immediately set to 0

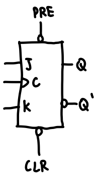
flip-flop works normally when PRE = CLR = not active ← can use to initialise flip-flop

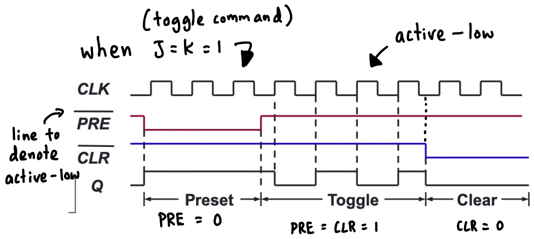
→ if PRE = CLR = active, invalid input!

* active-high: active = 1

* active-low: active = 0







State Table

m flip-flops + n inputs → 2^{m+n} rows

Present State		Input	Next State		Output
A	B	x	A^+	B^+	y
⋮	⋮	⋮	⋮	⋮	⋮

Compact →

Present State	Next State		Output	
	x=0	x=1	x=0	x=1
AB	A^+B^+	A^+B^+	y	y
⋮	⋮	⋮	⋮	⋮

State Diagram

state → circle

arrow → transition (with a label of I/O) ← input/output

m flip-flops → ≤ 2^m states

- circuit output functions
  → y = ⋯
- flip-flop input functions
  → JA = ⋯ } 1st letter: flip-flop input
  KA = ⋯ } 2nd letter: flip-flop name
- state equations
  A^+ = ⋯

Excitation Table

- used in design
- given transition, find flip-flop inputs

Q	Q^+	J	K	S	R	D	T
0	0	0	x	0	x	0	0
0	1	1	x	1	0	1	1
1	0	x	1	0	1	0	1
1	1	x	0	x	0	1	0
x	x	x	x	x	x	x	x

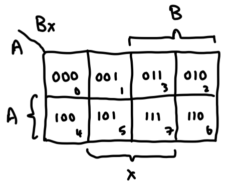
Analysis

Present State		Input	Next State		flip-flop inputs			
A	B	x	A^+	B^+	JA	KA	JB	KB
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

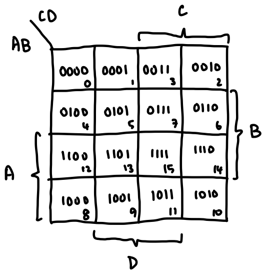
Start from circuit diagram, obtain state table or state diagram

if unused states, "x" for everything

draw K-map / truth table to get boolean functions

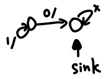


A \ Bx				
	000 0	001 1	011 3	010 2
A	100 4	101 5	111 7	110 6



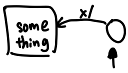
AB \ CD				
	0000 0	0001 1	0011 3	0010 2
	0100 4	0101 5	0111 7	0110 6
	1100 12	1101 13	1111 15	1110 14
	1000 8	1001 9	1011 11	1010 10

Sink : once the circuit enters this state, never exits



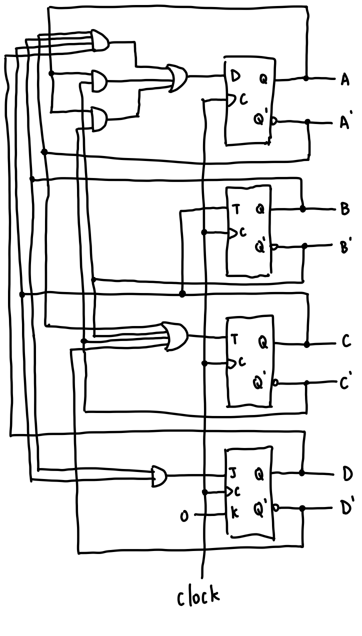
sink (does NOT have to be unused)

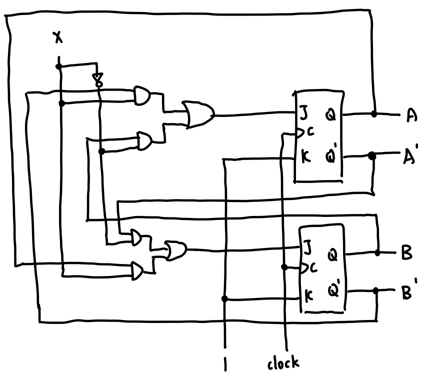
self-correcting circuit : if circuit (somehow) enters an unused state, will exit the unused state in a finite number of transitions



unused state

* draw slash even if only input/output present





Memory

→ stores program & data

1 byte = 8 bits

1 word: multiple of bytes, for transfer main memory ⇆ register
usually size of register

1 KB : 2^{10} bytes	powers of 2
1 MB : 2^{20} bytes	
1 GB : 2^{30} bytes	
1 TB : 2^{40} bytes	

desirable properties

- fast access
- large capacity
- economical cost
- non-volatile (saved even when no power)

} but most don't have all these

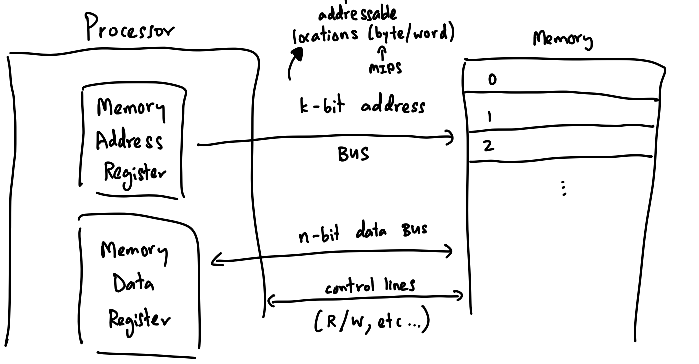
memory hierachy

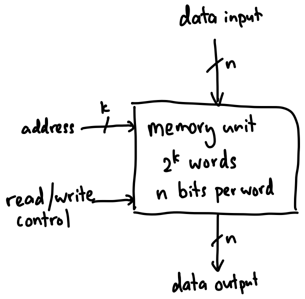
registers
main memory
disk storage
magnetic tapes

↑ fast, expensive (small numbers), volatile

↓ slow, cheap (large numbers), non-volatile

* up to 2^k addressable locations (byte/word)
↑ mips





memory enable	read/write	
0	x	none
1	0	write
1	1	read

Memory unit stores binary information in groups of bits (words)

data consists of n lines (for n-bit words)

data input lines carry data to be written

data output lines carry data to be read

address consists of k lines

control line read/write specifies direction of transfer

Write

- transfer address
- transfer data to be written
- set read/write = 0

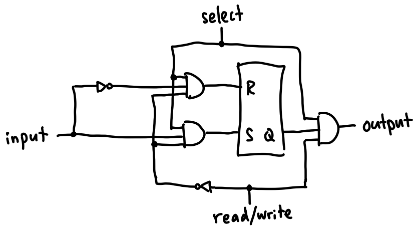
Read

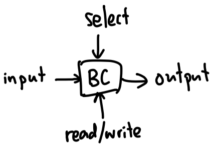
- transfer address
- set read/write = 1

RAM : Random Access Memory

Static RAM

- use flip-flops as memory cells



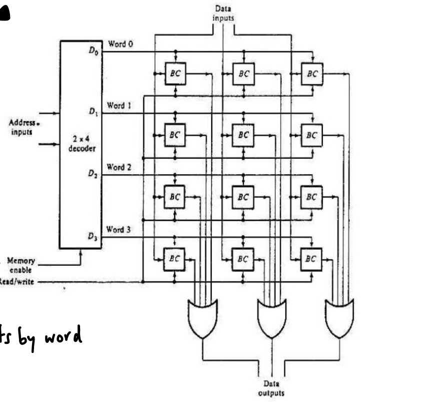


BC : binary/bit cell
[single bit]

Dynamic RAM

- use capacitor charges to represent data
- simpler in circuitry BUT need to be constantly refreshed

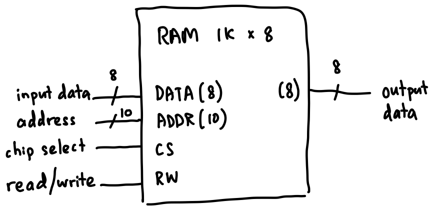
4x3 RAM



selects by word

→ keep combining blocks of RAM to make BIGGER blocks of RAM

word size
↓
1K x 8 : 1024 bits (2^{10})



4K x 8 : 4096 bits (2^{12} address)

2M x 32 : 2^{21} address, 32 bit word size
use 512k x 8 memory chips

2^{21} ∵ 2^{21} > 2 million

Pipelining

- does not help latency of single task
- helps throughput of entire workload

multiple tasks operating simultaneously using different resources

limited by:
- slowest pipelining stage
- stall for dependencies

MIPS Pipeline Stages

IF : instruction fetch

ID : instruction decode & register file read

Ex : execute / address calculation

MEM : memory access

WB : write back

Pipelined Implementation

- one cycle per pipeline stage
- data required for each stage needs to be stored separately because previous component might be used for something else already

Data used by subsequent instructions are stored in programmer-visible state elements : PC, register file, memory

Pipeline Registers

— not 32 bits (not really registers, its a collection of things)

IF/ID →	ID/EX →	EX/MEM →	MEM/WB →	~ end ~
· RR1 and RR2	· RD1 and RD2	· (PC + 4) + (immediate x 4)	· ALU result	· result is written back to register file (if applicable) using the write register stored in MEM/WB
· 16-bit offset (to be sign-extended to 32-bits)	· 32-bit immediate value	· ALU result	· memory read data	
· PC + 4	· PC + 4	· isZero? signal	· write register	
	· write register	· RD2		
		· write register		

✱ need to "pass" write register along the pipeline registers

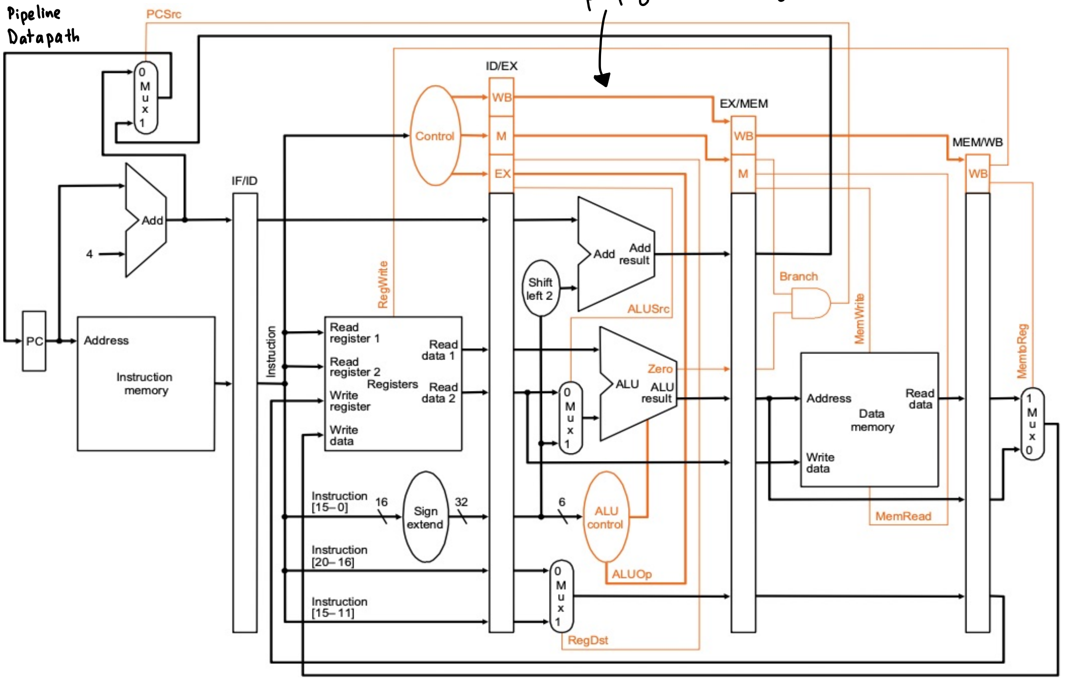
group control signals based on pipeline stage →

[IF & ID no control signals]

	EX Stage			MEM Stage			WB Stage	
	RegDst	ALUSrc	ALUOp	MemRead	MemWrite	Branch	MemToReg	RegWrite
R-type	1	0	10	0	0	0	0	1
lw	0	1	00	1	0	0	1	1
sw	x	1	00	0	1	0	x	0
beq	x	0	01	0	0	1	x	0

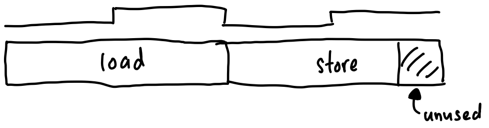
propogate control signal until utilised

Pipeline Datapath



Single-cycle Implementation

- update all state elements at the end of clock cycle (PC, register file, data memory)
- every instruction takes up 1 cycle
- cycle length based on slowest instruction

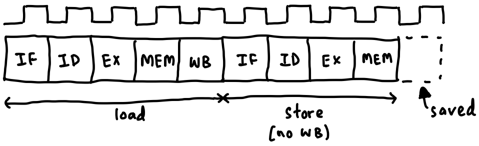


Cycle time: $CT_{seq} = \max\left(\Sigma_{k=1}^{N} T_k\right)$ ← bounded by slowest instruction

For j instructions: $Time_{seq} = j \times CT_{seq}$

Multi-cycle Implementation

- every stage takes up 1 cycle
- cycle length based on slowest stage
- instructions: n cycles ⟷ n stages



Cycle time: $CT_{multi} = \max(T_k)$ ← bounded by slowest stage

For j instructions: $Time_{multi} = \underbrace{j \times \text{average CPI}}_{\text{total cycles}} \times CT_{multi}$

Pipelining Implementation

Cycle time: $CT_{pipeline} = \max(T_k) + T_d$ ← overhead / latency ∵ pipeline register etc

For j instructions: $j + N - 1$ cycles ← after first, each additional only adds 1 more

$Time_{pipeline} = (j + N - 1) \times CT_{pipeline}$

Ideal case

assumptions
- every stage take equal time
- no overhead, $T_d = 0$
- $j >> N$

N: number of pipeline stages

$$
\begin{aligned}
\text{Speedup} &= \frac{Time_{seq}}{Time_{pipeline}} \quad \because \text{ratio} \\
&= \frac{j \times \Sigma_{k=1}^{N} T_k}{(j+N-1) \times (\max(T_k) + T_d)} \\
&= \frac{j \times N \times T_1}{(j+N-1) \times T_1} \quad \because T_1 = T_2 = \cdots = T_N \\
&\approx \frac{j \times N \times T_1}{j \times T_1} \quad \because j >> N \\
&= N \;\#
\end{aligned}
$$

Pipeline Hazards

Structural hazard
- simultaneous usage of hardware

Data hazard / dependency
- read / write data in same register

Control hazard / dependency
- to branch or not

} instruction dependencies

Solutions

1. Stall / delay pipeline
2. Separate Data and Instruction memory
3. Split register cycle into half
   - write then read

pipeline stage		hardware
IF	→	IM (instruction memory)
ID	→	REG
MEM	→	ALU
EX	→	DM (data memory)
WB	→	REG

separate to avoid structural hazard

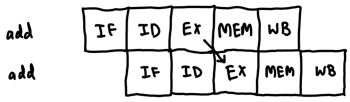
Read-After-Write (RAW) aka true data dependency

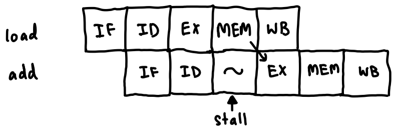
```
add $1,$2,$3
sub $4, $1, $2
```

← still uses stale/old value of $1

Soln: <u>Forward</u> the data to next instructions

Bypass data read from register file





[cannot solve with just forwarding, still need to stall]

WAR and WAW also exist but do <u>not</u> cause problems unless executed out of order

∵ conditional branching

Control Dependency

✱ can use all these methods in conjunction

① Stall pipeline

- waiting for branch outcome wastes 3 clock cycles [next instruction starts after <u>MEM</u> of beq/bne]
- branching happens very often so, not acceptable (BAD!)

② early branch resolution

magnitude comparator

- move computation of isZero? earlier [ID stage instead of MEM]
- reduced from 3 to 1 cycle delay
- but if register modified previously, still need to stall (RAW problem)

→ add	IF	ID	EX	MEM	WB		
beq		IF	~	ID	Ex	MEM	WB
next				~	IF	...	

⟷ 2

→ load	IF	ID	EX	MEM	WB			
beq		IF	~	~	ID	Ex	MEM	WB
next					~	IF	...	

⟷ 3 (same as naive)

③ branch prediction

- assume not branching & start computing
- flush successor instructions from pipeline (if branch) → terminate immediately

} simple/naive way

- flush happens before anything is written to register/memory so no problem
- if correct → no stall needed
- if fail → equivalent to not doing branch prediction in the first place

- software can optimise dynamic prediction based on previous branches (out of scope)

④ delayed branching (software/compiler solution)

- shift order of execution
- compute some non-control dependent instruction (known as branch-delay slot)
  ↳ for MIPS (w/ early branch), 1 slot

if no suitable instruction, use a NOP (no-op) instruction

~50% of the time can find

} done by compiler

✱easier with early branch resolution (less branch-delay slots to fill)

⑤ Multiple Issue Processor

not tested

- multiple instructions in each stage

static multiple issue

- compiler specifies set of instructions to be executed together
- simple hardware, complex compiler

Explicitly Parallel Instruction Compiler (EPIC)

Very Long Instruction Word (VLIW) — IA64

dynamic static issue

- hardware decides
- complex hardware, simple compiler

superscalar processor — most modern processors

Memory

accessing memory takes ~50x the clock cycles

	Registers	SRAM (uses flip-flop)	DRAM (main memory)	Hard Drive
transistors/memory cell		6	1	
density		low	high	
access latency	very fast 20ps	fast 0.5-5 ns	slow 50-70ns	very slow 5-20ms
cost	$$$$	$$$	$	¢
capacity	100s bytes	100s kB	100s MB	100s GB

ideal:

- small but fast memory near CPU
- large but small memory further away

DDR SDRAM

<u>D</u>ouble <u>D</u>ata <u>R</u>ate <u>S</u>ynchronous <u>D</u>ynamic <u>RAM</u>

↳ both positive & negative clock edges used

CS2100

make SLOW memory look FAST

- cache
- hardware managed

CS2106

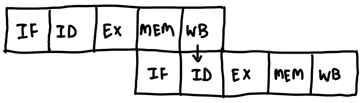
make SMALL memory look BIG

- virtual memory
- OS managed

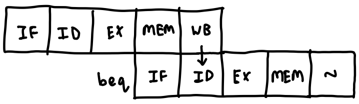
transparent to programmer

w/o forwarding & MEM stage

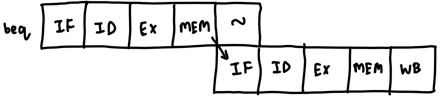
① RAW → +2



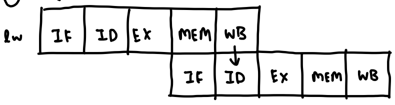
② RAW at beq → +2



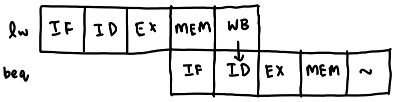
③ after beq → +3



④ lw → +2



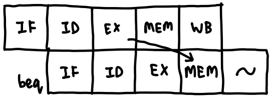
⑤ lw then branch → +2



w/ forwarding & MEM stage

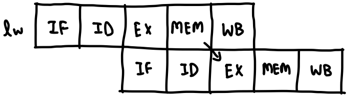
① RAW → +0

② RAW at beq → +0

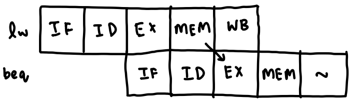


③ after beq → +3

④ lw → +1



⑤ lw then branch → +1



branch prediction

① if correct, after beq → +0

② if wrong, same as no branch prediction

jump instruction resolution

① at ID → +1

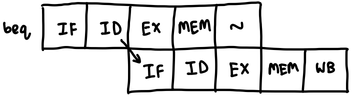
② at MEM → +3

w/o forwarding & ID stage

① RAW → +2

② RAW at beq → +2

③ after beq → +1



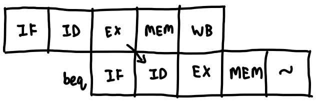
④ lw → +2

⑤ lw then branch → +2

w/ forwarding & ID stage

① RAW → +0

② RAW at beq → +1



③ after beq → +1

④ lw → +1

⑤ lw then branch → +2

☆ RAW can last for multiple instruction (w/o forwarding)

\$\$

Cache ← uses fast SRAM (usually) * also usable for instruction memory

Keep the frequently and recently used data in smaller but faster memory!

Principle of locality: Program accesses only a <u>small portion</u> of the memory address space within a <u>small time interval.</u>

<u>Temporal Locality</u>

If an item is referenced, tend to be referenced soon <u>again</u>.

<u>Spatial Locality</u>

If an item is referenced, <u>nearby items</u> tend to be referenced soon.

Instructions & data are in different localities!

Average Access Time = Hit Rate × Hit Time
+ (1 - Hit Rate) × Miss Penalty
↳ time to replace cache block + hit time ∴ miss penalty < hit time

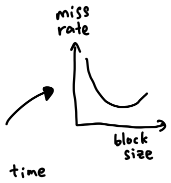
<u>larger</u> block size

++ spatial locality

-- larger miss penalty ∵ larger block to fill

-- lesser cache blocks → miss rate go up

→ need balance to maximise average access time



block size > word size
to exploit locality

Cache contains:

1. data block
2. Tag of memory block
3. Valid bit (initial: 0, after storing data: 1)

cache hit = valid bit is TRUE
&& tag[index] == tag in cache

<u>Compulsory Miss</u>

- on first access
- valid == FALSE
- aka cold start miss, first reference miss

<u>Conflict Miss</u>

- tag[index] ≠ tag in cache
- aka collision/interference miss
- FA cache no conflict misses

<u>Capacity Miss</u>

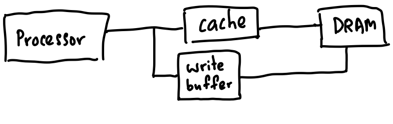
- blocks discarded from cache ∵ cache cannot contain all needed blocks
- occurs in FA Cache

→ on read miss:

1. load data into cache
2. load from there to register

Writing with a Cache

1. Write-through cache
   - write data both to cache and main memory
   - use write buffer to avoid slow write



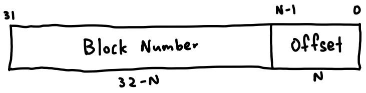
2. Write-back cache
   - only write to cache
   - only write to main memory when cache block is replaced/evicted
   - use extra bit (dirty bit) to store if a write occurred
     ↳ only write to main memory if dirty == TRUE

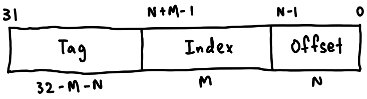
<u>Write Miss</u>

1. Write allocate
   - load the entire block into cache
   - change the word
   - write back depending on write policy

2. Write around
   - write directly to memory

Direct Mapping Cache





→ essentially 1-way SA cache

Number of blocks in memory = 2^{32-N}

total bytes (cache) = words × 4 (MIPS) = 2^{N+M} bytes ($\because 2^N \times 2^M$)

total bytes (block) = 2^N bytes (N = offset)

Number of cache blocks = 2^M (M = index)

Tag = 32 - (N+M) bits

↳ $2^{32-(N+M)}$ share same spot

Cache trashing

- 0% hit rate when A[i] and B[i] mapped to same cache location so constantly re-write to same block
- common reason: multiple of cache size

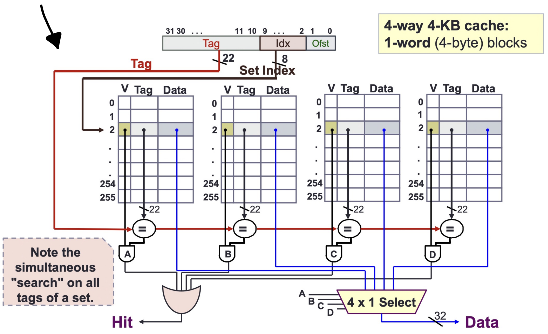
Set Associative (SA) Cache

← minimise conflict misses

N-way, N > 1, N = number of cache blocks in a set

1. find set (w/ set index)
2. place in any block in the set (prioritise left to right for CS2100)

∴ need to search all the cache blocks in the set (search is done concurrently)



4-way 4-KB cache:
1-word (4-byte) blocks

Note the simultaneous "search" on all tags of a set.

Miss rate of N-sized direct-mapped cache

≈ Miss rate of $\frac{N}{2}$-sized 2-way SA cache

memory addressing same as direct mapping

set index = block number % num of cache sets

↳ = $\frac{\text{num of cache blocks}}{N}$

NOT blocks!

number of cache sets = 2^M

(M = set index)

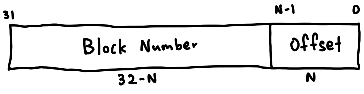
Fully Associative (FA) Cache

place memory block ANYWHERE (no more mapping function)

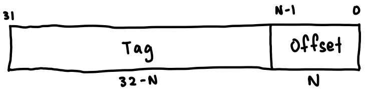
++ can be placed anywhere (no conflict miss)

-- need to search ALL cache blocks

- same cold miss
- capacity miss happens & decreases as cache size increases



block number = tag



total bytes (block) = 2^N bytes

(N = offset)

Tag = 32 - N bits

→ essentially all cache blocks in same set of SA cache

Block Replacement Policy

① Least Recently Used (LRU)

- for cache hit, record the cache block that was accessed
- when replacing, replace the one that has not been used for the longest time
- good ∵ temporal locality
- but can be hard to keep track of

LRU ↓   MRU ↓

0 4 8 12

↓ access 4

0 8 12 4

↓ access 12

0 8 4 12

↓ access 16

removed ↓

0 8 4 12 16

NOT a cache

[priority queue that tracks "least recently used"]

↑ one for each set

② FIFO queue

③ Random replacement (RR)

④ Least frequently used (LFU)

Multilevel Cache

- use multiple caches (w/ different associativities) in tandem!
- Modern CPUs have 3 levels: L1, L2, L3

↑ Intel/AMD

			2	
	0	1	3	2
1	4	5	7	6
		3		

			2	
	0	1	3	2
1	4	5	7	6
		3		

			2	
	0	1	3	2
1	4	5	7	6
		3		

			2	
	0	1	3	2
1	4	5	7	6
		3		

			2	
	0	1	3	2
1	4	5	7	6
		3		

			2	
	0	1	3	2
1	4	5	7	6
		3		

			3		
	0	1	3	2	
	4	5	7	6	2
1	12	13	15	14	
	8	9	11	10	
		4			

			3		
	0	1	3	2	
	4	5	7	6	2
1	12	13	15	14	
	8	9	11	10	
		4			

			3		
	0	1	3	2	
	4	5	7	6	2
1	12	13	15	14	
	8	9	11	10	
		4			

			3		
	0	1	3	2	
	4	5	7	6	2
1	12	13	15	14	
	8	9	11	10	
		4			

			3		
	0	1	3	2	
	4	5	7	6	2
1	12	13	15	14	
	8	9	11	10	
		4			

			3		
	0	1	3	2	
	4	5	7	6	2
1	12	13	15	14	
	8	9	11	10	
		4			

0	0	0	0	0			
1	0	0	0	1			
2	0	0	1	0			
3	0	0	1	1			
4	0	1	0	0			
5	0	1	0	1			
6	0	1	1	0			
7	0	1	1	1			
8	1	0	0	0			
9	1	0	0	1			
10	1	0	1	0			
11	1	0	1	1			
12	1	1	0	0			
13	1	1	0	1			
14	1	1	1	0			
15	1	1	1	1			

unused = x

0	0000	
1	0001	
2	0010	
3	0011	
4	0100	
5	0101	
6	0110	
7	0111	
8	1000	
9	1001	
10	1010	
11	1011	
12	1100	
13	1101	
14	1110	
15	1111	

0	0000	
1	0001	
2	0010	
3	0011	
4	0100	
5	0101	
6	0110	
7	0111	
8	1000	
9	1001	
10	1010	
11	1011	
12	1100	
13	1101	
14	1110	
15	1111	

	ABCD	AB	AC	AD	BC	BD	CD
0	0000	00	00	00	00	00	00
1	0001	00	00	01	00	01	01
2	0010	00	01	00	01	00	10
3	0011	00	01	01	01	01	11
4	0100	01	00	00	10	10	00
5	0101	01	00	01	10	11	01
6	0110	01	01	00	11	10	10
7	0111	01	01	01	11	11	11
8	1000	10	10	10	00	00	00
9	1001	10	10	11	00	01	01
10	1010	10	11	10	01	00	10
11	1011	10	11	11	01	01	11
12	1100	11	10	10	10	10	00
13	1101	11	10	11	10	11	01
14	1110	11	11	10	11	10	10
15	1111	11	11	11	11	11	11

	ABCD	AB	AC	AD	BC	BD	CD
0	0000	00	00	00	00	00	00
1	0001	00	00	01	00	01	01
2	0010	00	01	00	01	00	10
3	0011	00	01	01	01	01	11
4	0100	01	00	00	10	10	00
5	0101	01	00	01	10	11	01
6	0110	01	01	00	11	10	10
7	0111	01	01	01	11	11	11
8	1000	10	10	10	00	00	00
9	1001	10	10	11	00	01	01
10	1010	10	11	10	01	00	10
11	1011	10	11	11	01	01	11
12	1100	11	10	10	10	10	00
13	1101	11	10	11	10	11	01
14	1110	11	11	10	11	10	10
15	1111	11	11	11	11	11	11

Fast boolean algebra

2: 3, 5

3: F, 33, 55

4: FF, F0F, 3333, 5555

5: FFFF, FF00FF, F0F0F0F, 3333 3333, 5555 5555