1. Data representation and number systems

① Number systems

1) radix systems : weight in powers of R

1. base 10 (decimal) ⟶ $12345 = 5\times10^0 + 4\times10^1 \dots 1\times10^4$

2. base 2 (binary) ⟶ $1101 = 1\times2^0 + 1\times2^2 + 1\times2^3$

3. base 16 (hexadecimal)

2) notation (prefixes) $\underline{0}32 \Rightarrow$ octal ; $\underline{0x}32 \Rightarrow$ hexadecimal ; 0b ⇒ binary

3) conversion between bases of diff radix

base R ⟷ decimal ⟷ binary ⟷ octal ⟷ hexadecimal

(groups of 3 between binary and octal; pairs between octal and hexadecimal; groups of 4 between binary and hexadecimal)

whole numbers : repeated division till quotient is 0

fractions : repeated multiplication till fractional product is 0 to sufficient dp

eg. $0.3125_{10} = 0.0101_2$

$0.3125 \times 2 = 0.625$

$0.625 \times 2 = 1.25$

$0.25 \times 2 = 0.50$

$0.50 \times 2 = 1.00$

② Data representation in binary

1) terminology
- nibble : 4 bits
- byte : 8 bits
- word : multiple of bytes, depending on computer architecture

2) range and overflow

↳ N bits can represent up to 2^N values ⟶ add to many, will overflow max ⇌ min (+1 / −1)

↳ $\log_2 N$ bits to represent N value

↳ $\frac{\lg N}{\lg 2}$

③ ASCII and unicode : representing characters

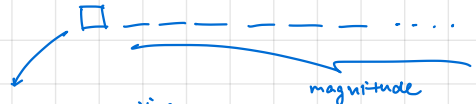
ASCII: 8 bits, 128 char ; unicode: 32 bits, 4 billion char

note: in C, int and char are 'somewhat interchangeable'.
if value is int from 0–127 but evaluated to char, will return according to ASCII

(4) Representing negative numbers in binary

1) **sign and magnitude** → first bit used for sign

↳ how to read



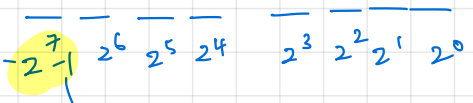
1, negative; 0, positive
to negate, just flip

range : $2^{N-1} \dots 1, 0, -0, -1 \dots -2^{N-1}$ (not very useful)

$\pm(2^{N-1}-1)$

2) **1s complement**

↳ how to read :



note that 1111 ... is -0, 0000... is +0

1 indicates negative, 0 positive

↳ range : $\pm(2^{N-1}-1)$

largest	0 111 1111	+127
smallest	1 000 000	-127
zeros	0 000 0000	+0
	1 111 1111	-0

↳ how to obtain

$$-x = 2^n - x - 1$$

$-x = r^{(n)} - r^{(m)} - x$ (more generally)

n = no. integer digits; m = no of fractional digits

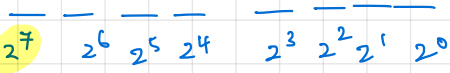
(R-1)'s complement (radix (base))

eg. For 8 bit number : $-12 = 2^8 - 12 - 1$

$= 243$ → convert to binary → 1111 0011

↳ to negate, simply invert all bits (binary)

3) **2s complement** (indicates positive/negative)



↳ how to read :

↳ range: -2^{n-1} to $2^{n-1}-1$

max	0111 1111	127
min	1000 0000	-128
zero	0000 0000	0

↳ how to obtain :

$$-x = 2^n - x$$

$-x = R^n - x$ (R's complement) ⇒ convert to binary

(gets its name from the rule that sum of n-bit number and negative is R^n.)

↳ to negate, invert all bits, binary addition +1

4) complement fractional number : just do as per normal

5) **excess representation**

↳ how to read : convert binary to decimal − excess (excess = 2^{N-1})

6) **sign extension**

1110 = 11111110 , 1100 = 11 00.00000

↳ copy MSB to extend leftwards, or copy LSB to extend rightwards.

(technically padding, not sign extension)

7) comparison

1. when positive, sign/magnitude | 1s comp | 2s comp are identical.
2. when negative, need to note difference.

⑤ binary operations

1) addition & subtraction :

$1110 + 1 = 1111$; $0111 + 1 = 1000$; $1111 + 1 = 0000$ overflow!

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

2s complement

algorithm for addition :
binary addition, discard end carry

1s complement

algorithm for addition :
binary addition, add 1 to result if carry out from msb

detecting overflows : positive + positive → negative
negative + negative → positive

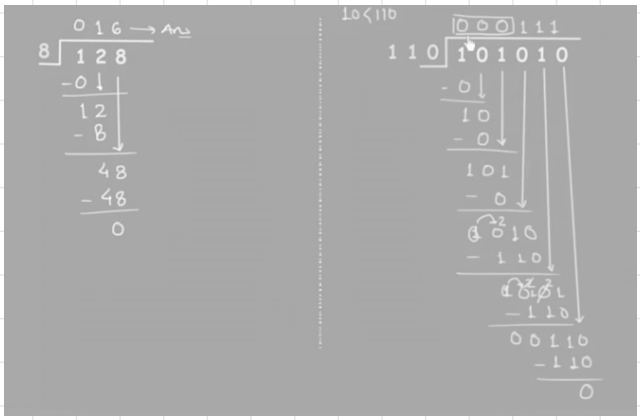
2) multiplication

$0 \times 0 \Rightarrow 0$
$1 \times 0 \Rightarrow 0$
$1 \times 1 \Rightarrow 1$

eg. 1001×0101

```
  1001
x 0101
------
  1001
 00000   <- indent everytime
100100
+0000000
--------
 101101
```

3) division



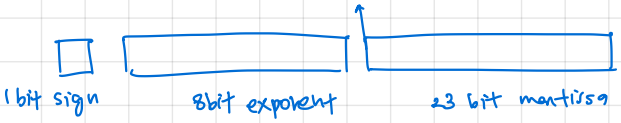
⑥ floating point representation (IEEE 754)

↳ single precision format (32 bits) : principle based on scientific notation

eg. $110.1_2 = 1.101_2 \times 2^2$

$120 = 1.20 \times 10^2$

leads to an implicit 1



8bit exponent : in excess 127 notation → magnitude - 127

23 bit mantissa : effectively 24 bit unsigned (in base 2)

using biased excess exponent there are two bit patterns that do not "mean anything"

1111 1111 ... and 0000 000 ...

- 1111 1111 ... : pattern used for ∞
- 0000 000 ... : indicates implicit 0 rather than implicit 1 for mantissa

↓

allows representation of 0

AND smallest number, because when exponent is 0 :

$0.1000000000 \times 2^{-126} = 2^{-127}$ because -126 is rather than -127

⇒ storage of 0.0 is a special case where if exponent field is all zeros then implied 1 is not added to mantissa

1.11111111

Notes

1.

Hexadecimal	Binary	Decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

2. proof of sign extension for 2s complement from n bit to m bit

1. if positive, 0 makes no diff
2. if negative:

$$-2^m + \underbrace{2^{m-1} + \dots + 2^{m-n}}_{m-n \text{ terms, geometric progression}} + 2^n + 2^{n-1} \dots = -2^m + \frac{2^n(2^{m-n}-1)}{2-1} + 2^{n-1} \dots$$

$$= -2^m + 2^m - 2^n + 2^{n-1} \dots$$

$$= \text{x}$$

3. be careful in binary addition. Sign extend first, then add

4. 0.1 cannot be exactly represented in binary

2. Introduction to C

① Fundamentals

1) declaration ⟶ block of memory allocated to variable, but uninitialized variables contain random value

```
int count;
int count = 4
```

2) initialisation ⟶ variable assigned an actual value

3) types

↳ int : 4 bytes, −2 147 483 648 (-2^{31}) to 2 147 483 647 ($2^{31}-1$)

↳ float : 4 bytes

↳ double : 8 bytes

↳ char : single quotes, 1 byte

② program structure

1) preprocessor directives

1. inclusion of header files — `#include <  .h >`

↳ to use function from libraries and let compiler know to refer these at compilation. `gcc -lm`

↳ useful libraries : stdio.h, string.h

General form

```
preprocessor directives
main function header
{
    declaration of variables
    executable statements
}
```

"Executable statements" usually consists of 3 parts:
- Input data
- Computation
- Output results

2. macro expansions

↳ tells compiler to substitute all appearances of this at compile time. Do not use;

e.g. `#define PI 3.142`

2) input/output

1. input — `scanf("%d, %f", &age, &cap)` (format; addresses to store inputs)

2. output — `printf("string")` or `printf("Hi %d", age)` (string; what to get variable value for placeholder)

Placeholder	Variable Type	Function Use
%c	char	printf / scanf
%d	int	printf / scanf
%f	float or double	printf
%f	float	scanf
%lf	double	scanf
%e	float or double	printf (for scientific notation)

Python: All inputs a… read as `string`

- Examples of format specifiers used in printf():
  - %5d: to display an integer in a width of 5, right justified
  - %8.3f: to display a real number (float or double) in a width of 8, with 3 decimal places, right justified

3) computation / execution : notes

1. variable assignment
   - ↳ assignment also returns value of its right side expression → hence
   - ↳ can be cascaded from right to left `q = (b = c = 3+6);`

2. results that cannot be stored are truncated

3. operator precedence

Operator Type	Operator	Associativity
Primary expression operators	`( ) expr++ expr--`	Left to right
Unary operators	`* & + - ++expr --expr (typecast)`	Right to left
Binary operators	`* / %`	Left to right
	`+ -`	
Assignment operators	`= += -= *= /= %=`	Right to left

→ return value then do
→ increment first then return
→ do then return
→ `x += y += 5` ⇒ `y = y+5` `x = x+y`

4. mixed type operation: result depends on type declaration and value used during computation

```
int   m = 10/4;      means   m = 2;
float p = 10/4;      means   p = 2.0;
int   n = 10/4.0;    means   n = 2;
float q = 10/4.0;    means   q = 2.5;
int   r = -10/4.0;   means   r = -2;   Caution!
```

5. % gives remainder, not modulo

Python % is modulo
```
a = 10%4  → a = 2
b = -10%4 → b = 2
```
C % is remainder
```
a = 10%4  → a = 2
b = -10%4 → b = -2
```

④ control structure

↳ no boolean types in C ⇒ use integers. 0 → false, non-0 → true

1) evaluation of boolean expressions

↳ short circuit evaluation

Operator Type	Operator	Associativity
Primary expression operators	`( ) [] . -> expr++ expr--`	Left to Right
Unary operators	`* & + - ! ~ ++expr --expr (typecast) sizeof`	Right to Left
Binary operators	`* / %`	Left to Right
	`+ -`	
	`< > <= >=`	
	`== !=`	
	`&&`	
	`||`	
Ternary operator	`?:`	Right to Left
Assignment operators	`= += -= *= /= %=`	Right to Left

Python
```
cond ? expr1 : expr2 →
expr1 if cond else cond2
```

switch - case - default

```
for ( )
switch ( ) {
    case 0 : ( )
    case -2 : break
    default :
}
```

next loop

↓ spill

2) loops

while

```
while ( condition )
{
   // loop body
}
```

do-while

↳ do atleast once

```
do
{
   // loop body
} while ( condition );
```

for

```
for ( initialization; condition; update )
{
    // loop body
}
```

Initialization: initialize the **loop variable**

Condition: repeat loop while the condition on **loop variable** is true

Update: change value of **loop variable**

optional. for (;;) => infinite loop

use of 'continue' and 'break'

skip to next iteration

out of loop

(3) Pointers

1) pointers

↳ address of a variable v: &v , i/p format specifier

↳ addresses printed in hexadecimals

↳ address varies from run to run as system allocates any free memory to the variable

↳ pointer variable : variable that contains address of another variable

2) declaration and initialisation

```
type * pointer_name ;
type * pointer_name = &a ;
```

points to random address if not initialised

3) dereferencing

```
int *a_ptr = &a
b = *a_ptr
```

assigning value of b to pointer variable a_ptr

```
a_ptr = b
```

dereferencing → *a_ptr is synonymous to a

4) incrementing pointers

```
type *a_ptr = &a ;
a* ptr ++ ;
```

depending on type, incrementing will change address by no. of bytes of memory address pointer is pointing at.

- If p is a pointer variable, what does p = p + 1 (or p++) mean?

Recall Lect#2 slide 15:
int takes up 4 bytes
float takes up 4 bytes
char takes up 1 byte
double takes up 8 bytes

```
int a; float b; char c; double d;
int *ap; float *bp;
char *cp; double *dp;

ap = &a; bp = &b; cp = &c; dp = &d;
printf("%p %p %p %p\n", ap, bp, cp, dp);
```

ffbff0a4 ffbff0a0 ffbff09f ffbff090

```
ap++; bp++; cp++; dp++;
printf("%p %p %p %p\n", ap, bp, cp, dp);
```

ffbff0a8 ffbff0a4 ffbff0a0 ffbff098

```
ap += 3;
printf("%p\n", ap);
```

ffbff0b4

IncrementPointers.c

⑥ Functions

```
$ gcc -lm MathFunctions.c
$ a.out
Enter x and y: 3 4
pow(3,4) = 81.000000
Enter value: 65.4
sqrt(65.400002) = 8.087027
```

1) calling functions
   - ↳ preprocessor directives, function prototype, linking at compile time
     - ↳ type name (type 1, type 2 ...);
     - without function prototype, compiler assumes default return type of int for function → conflict if actual function different

2) scope rule
   - ↳ parameters and variables declared within function are local → only accessible in the function they are declared (scope rule / lexical scoping)

3) use of pointers ← unable to modify values outside of function + unable to do multiple return

type function (type *p1 , ...)

When a function is called, an activation record is created in the call stack, and memory is allocated for the local parameters and variables of the function.

Once the function is done, the activation record is removed, and memory allocated for the local parameters and variables is released.

Hence, local parameters and variables of a function exist in memory only during the execution of the function. They are called automatic variables.

In contrast, static variables exist in the memory even after the function is executed. ↳ ie. can only use within function, but access multiple function calls

⑦ Arrays

- ↳ homogeneous collection of data
- ↳ elements occupy contiguous memory location and are accessed through indexing

1) declaration and initialisation
   - type name [size]; → array elements initialised to random value
   - type name [] = { 1, 2, 3 }
   - type name [size] = { 1, 2, 3 }
     - ↳ if no. of elements < size, remaining elements initialised to 0
   - ↳ cannot initialise after declaration : only assign

2) arrays and pointers
   - ↳ array variable name is a fixed pointer to a [0] ⇒ cannot be altered

   eg.
   ```
   int a[] = {1,2}
   int b[] = {3,4}
   a=b //error
   ```

3) assignment
   - ↳ use loops, since direct variables not allowed

4) arrays as function parameters
   1. declaration in prototype and function definition
      - type[] enough. no need name, but good practice.
      - type name[] enough. No need size, but useful.
      - alternatively, *arr also allowed.

5) size of an array

⑧ Strings

↳ we can turn an array of characters into a string by adding '\0' null character at end of array
↳ a string is a character array terminated by '\0' → ASCII value of 0

1) declaration and initialisation

declare, initialisation by assignment — directly

```
char arr[size];
arr[0] = 'A';
   :
char[n] = '\0'
```

without, is just a char[], not a string

```
char arr[] = "string"
char arr[] = { 'a', 'b', ... '\0' }
```

2) input and output

input

Read string from stdin (keyboard)

```
fgets(str, size, stdin) // reads size - 1 char,
                        // or until newline
scanf("%s", str);  // reads until white space
```

output

Print string to stdout (monitor)

```
puts(str);  // terminates with newline
printf("%s\n", str);
```

3) string functions from #include <string.h>

- strlen(s) : no. of char
- strcmp(s1, s2) : → 0 (identical)
- strcmp(s1, s2, n) → difference in ASCII value if not same for unmatched character from left to right. s1[n] − s2[n]
  ↳ compare first n characters
  - negative: s1[n] < s2[n]
  - positive: s1[n] > s2[n]
- strcpy(s1, s2)
  ↳ copy string s2 into s1 location, returns s1
- strcpy(s1, s2, n)
  ↳ copy first n characters of string pointed to by s2 to s1

4) importance of '\0' character

↳ %s and string functions only work on true strings, often because they iterate through char[] until '\0'
eg. printf("%s"), strlen() ⇒ may result in segmentation fault

⑨ structures

↳ grouping of heterogeneous data (subject to word alignment) in the order they are declared

↳ elements occupy contiguous memory location and are accessed through dot operator

1) defining structure types

```
typedef struct {
    type name1;
    type name2;
    :
} name_t
```

2) declaration and initialisation : ala arrays

defined_type name = { all fields }

defined_type name; — fields are randomly initialised, as normal variables would be at declaration

- Examples:

```
typedef struct {
   int day, month, year;
} date_t;

typedef struct {
   int cardNum;
   date_t expiryDate;
} card_t;

card_t card1 = {888888, {31, 12, 2020}};
```

```
typedef struct {
   int stuNum;
   float score;
   char grade;
} result_t;

result_t result1 = { 123321, 93.5, 'A' };
```

3) reading and modifying structure members

struct_variable.field ⇒ treat as an ordinary variable

4) structures and pointers

↳ structure variable points to first field in structure struct struct1 = { ... }

↳ pointer is not immutable/fixed, can be assigned struct struct2 = struct1 // ok

5) structures and functions

output

↳ put struct_type as return type

↳ declare internal struct variable

↳ var.field to access/modify field

input

default - pass by value — pass by reference

↳ structure is copied into (shallow) function → if pointers inside, then will just copy reference, since unlike array, not fixed

↳ does not actually modify original structure variable

note that if array of structures passed in, then will modify actual

↳ pass by pointer

6) The arrow operator

(*struct_pointer).field ⇒ same as structpointer -> field

↳ note that diff from *(struct_pointer.field), since dot higher precedence than * (equivalent)

(10) compilation process

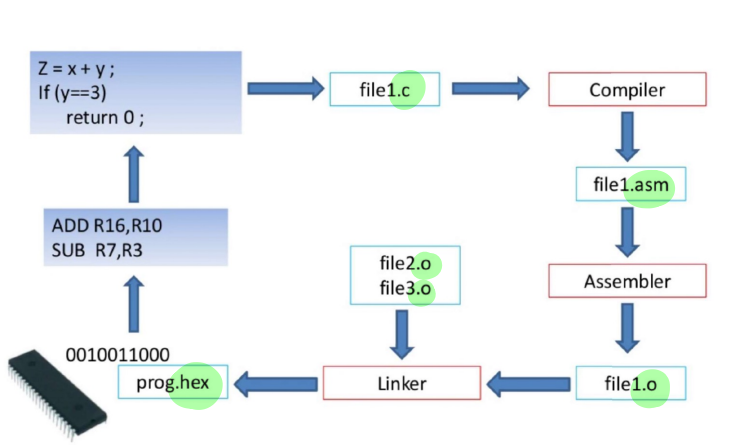
1) preprocessing : processes preprocessor directives like include-files, conditional compilation instructions and macros

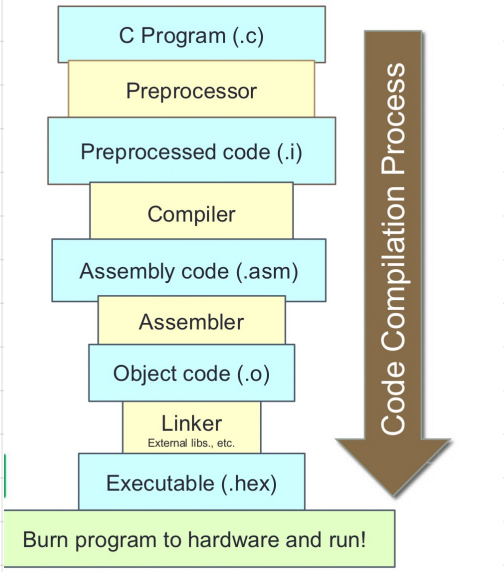
2) compilation : takes preprocessed code and generates assembly file

3) assembler : takes assembly code and generates object code

4) linking : takes one or more object files or libraries and linker script as input, combines them to generate a single executable file

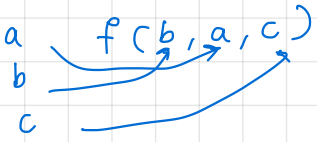
↳ resolves references to external symbols, assigns final addresses to procedures and variables. Revises data and code to reflect new addresses (known as relocation) ⇒ run





Notes

1. be careful of what is actually passed into a function eg.



3. Assembly and computer operations

(1) instruction set architecture

1) what is ISA

↳ set of formalisations to define how to encode instructions in binary so that hardware can execute program

↳ abstractions allow programmers to make machine code work as intended

2) machine code vs. assembly language

machine code: in binary, not human readable

assembly language: ↳ low level language that translates to machine code

3) registers

↳ high-speed memory storage in the processor

↳ typical ISA has 16 – 32 registers — more, easier for more complex instructions, but slower

↳ compiler associates variables in program / memory in registers and allows for operation on them

↳ registers have no data type ⇒ instructions determine how bits are interpreted (see encoding; chip is built based on encoding)

MIPS 32 registers, 32 bits each

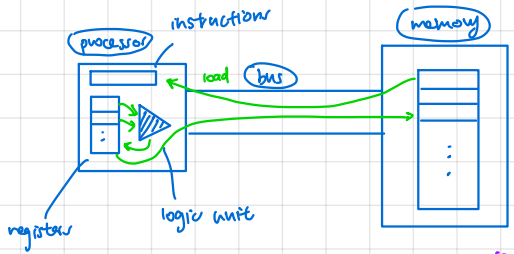
Name	Register number	Usage
$zero	0	Constant value 0
$v0-$v1	2-3	Values for results and expression evaluation
$a0-$a3	4-7	Arguments
$t0-$t7	8-15	Temporaries
$s0-$s7	16-23	Program variables

Name	Register number	Usage
$t8-$t9	24-25	More temporaries
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address

$at (register 1) is reserved for the assembler.
$k0-$k1 (registers 26-27) are reserved for the operation system.

4) how a chip works to run a program (the load store model)

↳ relying on registers for storage during execution



memory is a large, single dimensional array with address (in binary) acting as index, starting at 0.

1. source code and data reside in memory, loaded into processor; instructions to instructions, variables etc. to registers, through bus. (load)
2. operations on registers: sometimes using constant (immediate) values rather than from a register. (calculation)
3. instructions sequentially loaded and executed from memory, controlled by branching (relative addressing)
4. values moved back from registers to memory (store)

② MIPS assembly language (calculation)

1) general syntax

↳ each line at most 1 instruction

↳ # comment

involving only registers | involving memory

```
add  $s0, $s1, $s2
```

Operation (op) | Destination (gets the result) | Source 1 | Source 2

```
$s0 = $s1 + $s2
```

2) arithmetic operations add, sub, addi, subi

3) immediate operations

↳ immediate values are numerical constants that are often used

↳ MIPS supplies a set of operations specifically for them

↳ MIPS uses (16) bit immediate values, (2s) complement

(16) because of encoding; (2s) because largest range

4) logical operations sll, srl, and, andi, or, ori, nor, xor

⊕ ⇒ ~(or( ))

↳ multiply divide by 2^n, removes values / sign extension as bits shifted

sequential bitwise comparison

useful notes

- and: x and 0 = 0; x and 1 = x; x and x = 1
- or: x or 1 = 1; x or 0 = x
- xor: x xor x = 0; x xor 0 = x; x xor 1 = flipped
- nor: x nor 1 = 0; x nor x = flip; x nor 0 = flip

copy to temp andi temp, $1, 0010 0001

clear target positions

andi $2 $2, 1110 0111

paste

or $2 $2, temp

flip and copy to temp

```
xor  temp , $1 ,  0001 0001  #flip
andi temp , temp , 0001 0001
      # hide positions of non interest
```

5) loading constants ? 16 bits

1. use lui to load top 16 bits lui $t0, 0x...
2. ori to load in bottom 16 bits ori $t0, 0x...

6) complex expressions

↳ a single MIPS instruction can handle at most two source operands → need to break complex into multiple MIPS

(3) memory organisation and MIPS instructions (load/store)

1) memory organisation

↳ main memory can be viewed as a large, single-dimensional array of memory locations

↳ given a k-bit address, the address space is of size 2^k

2) transfer units and memory access

↳ using distinct memory addresses, we can access
- → a single byte (byte addressable)
- → a single (word) (word addressable)
  - ↳ usually 2^2 bytes. MIPS is $2^2 = 4$ bytes, 32 bits. commonly equal to register, integer and instructions in most ISAs

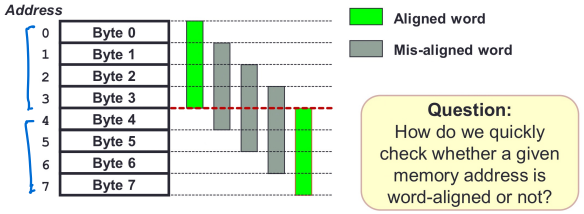
3) word alignment

↳ words are aligned in memory if they begin at a byte address that is a multiple of no. of bytes in word.

↳ to check: see if address divisible by 2^k

↳ ISAs like MIPS cannot load words that are unaligned.
because very slow, need to load both words, shift, store

Example: If a word consists of 4 bytes, then:



Question:
How do we quickly check whether a given memory address is word-aligned or not?

4) MIPS memory instructions

load word: `lw $t0, 40($s3)`

store word: `sw $t0 0($s3)`

load byte: `lb $t0 13($s3)`

store byte: `sb $t0 (1)($s3)`

offsets in multiples of word size

address

no need offsets of multiples of no. of bytes in word

5) other memory instructions

pseudo instructions

ulw, usw (unaligned load/store)

lh, sh, lwl, lwr, swl, swr

6) byte vs. word

↳ if machine does byte addressing, then consecutive words in memory have addresses that differ by 2^k bytes

↳ if word addressing, then differ by 1.

④ control structures in MIPS

1) branches

conditional

```
bne $t0, $t1, label
beq $t0, $t1, label
```

unconditional

```
jump label
```

2) inequalities

```
slt $t0, $s1, $s2
```

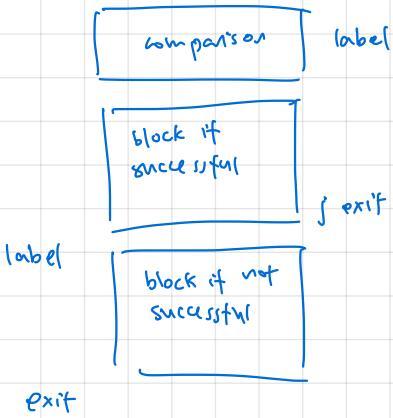
```
if ($s1 < $s2)
    $t0 = 1;
else
    $t0 = 0;
```

```
slt $t0, $s1, $s2
bne $t0, $zero, L
```

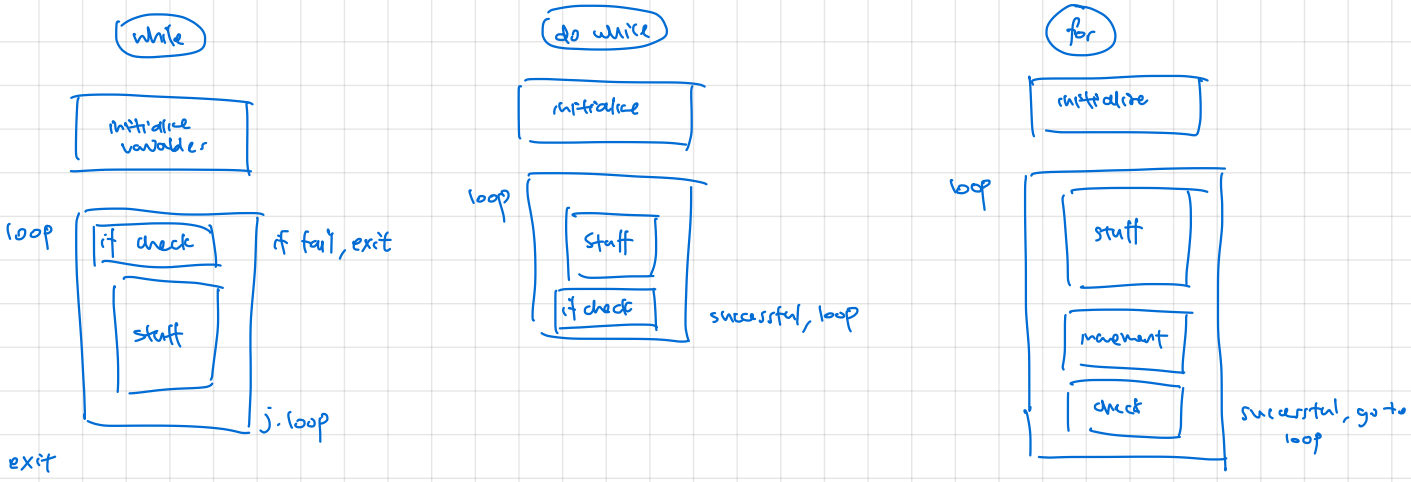
==

```
if ($s1 < $s2)
    goto L;
```

3) if - else



4) while, for, do-while loops



5) looping through arrays

pointer

- movement pointer register
  - ↳ addi pointer, pointer, 4

indexing

- increment index, add pointer + index x 4
  - ↳ addi $t0, index, 1
  - ↳ sll $t0, $t0, 2 (x4)
  - ↳ add pointer, pointer, $t0

⑤ Instruction addressing in MIPS

1) Instruction addresses in MIPS

↳ As instructions are stored in memory, they also have addresses

↳ Since instructions are also 32 bits, they are word aligned → branches, jumps only to word-aligned addresses

↳ PC (program counter): a special register that keeps address of instruction to be executed next in the processor

2) ways to encode memory location of thing of interest

register addressing

↳ operand is stored in register, specified by 5-bit address

eg. add $s0, $s1, $s2

immediate addressing

↳ operand is stored in instruction itself

eg. addi $s0 $s1 imm

base-relative addressing

↳ operand is at location whose memory address is the sum of a register (base) and immediate value

↳ eg. lw, sw $s0, imm($s1)

PC-relative addressing

↳ operand (instruction) is stored at memory address that is sum of PC and immediate value

eg. bne, beq $s0, $s1, immediate

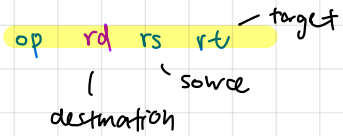
pseudo-direct addressing

↳ operand (instruction) is stored at memory address that is described by upper-4 bits of PC concatenated w 26 bits and 2 00 bits at end.

⑥ MIPS instruction encoding

↳ each MIPS instruction has a fixed length of 32 bits

1) Register (R) format



↳ partially specifies instruction

↳ equal to 0 for all R format instructions

registers of operand

↳ register that will receive result of computation

↳ amount a shift instruction will shift by. 0-31.

↳ set 0 in all non-shift instructions

↳ combined w opcode to exactly specify instruction

2) Immediate (I) format

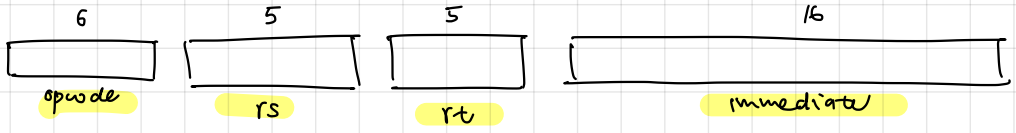
byte addressed, so +4 is next (one word)

1. General (addi, lw)

target to store

op rt rs immediate (OR) op rt (immediate) rs

source



↳ no funct field. opcode uniquely defines instruction.

↳ source register operand

↳ register to receive result

↳ 16 bit 2's complement signed integer

↳ up to 2^{16} diff values

↳ range: -2^{15} — $2^{15}-1$

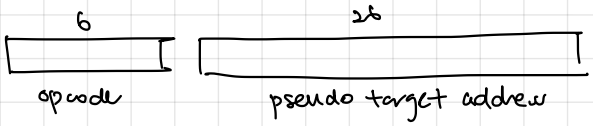
can overflow PC

2. Instruction access (branching) eg. beq, bne (order of encoding follows instruction)

↳ PC relative addressing

↳ 16 bits immediate value → interpret as words → word addressed (so +1 is 1 line), labels don't count → can branch to -2^{17} — $2^{17}-1$ instructions away from PC

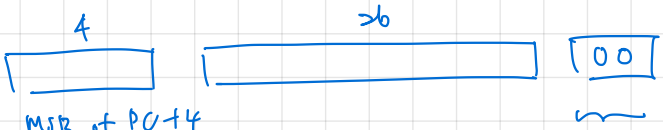
3) J-format (jump)



↳ to encode jump instruction address with 26 bits:

jump is to absolute address

so to jump from current PC → can just copy current PC pattern and adjust if needed



↳ if wants to jump to desired instruction, then multiple jumps

↳ range of jump: 2^{28} → $2^3 \times 2^{10} \times 2^{10} \Rightarrow$ 256 MB $\Rightarrow$ 2^{26} lines (words)

↳ can combine beq/bne w j to do long range branching

Notes

1. be careful of address vs content@address
   address → add
   content@address → lw, sw

2. operation in ALU during beq is subtraction!

3. MIPS complex operations: to get shortest, factorise out, use X and +
   eg. $3x = 4x - x$
   vs. $x + x + x$

4. when dealing w strings or character arrays:
   lb (load byte) — pointer increment by 1 byte!

5. if OR/AND is used → write value out. often used for masking.
   15/F 1111
   7 0111
   3 0011
   1 0001

6. editing array: a[i] = x+2 - lw
   add
   sw ⇒ don't forget

7. To change binary value in single instruction: desired − difference = current
   add difference

4. Instruction set architecture

① overview of ISA design

1) key choices in ISA design

1. Data storage architecture : where / how operands are stored and retrieved
2. memory addressing : endianness and how memory locations are encoded
3. operations in the instruction set
4. Encoding instruction set and instruction formats : length & fields

2) RISC vs CISC machines

1. CISC : complex instruction set computer eg. Intel x86-32
   - ↳ single instruction performs complex operations stored as microcode
   - ↳ smaller program size
   - ↳ complex implementation, no room for hardware optimization

2. RISC : reduced instruction set computer eg. MIPS, ARM
   - ↳ instruction set kept small and simple, hardware easier to implement and optimize
   - ↳ burden on software (eg. compiler, assembler) to combine operations to implement high-level language statements

② Data storage architecture

↳ von Neumann architecture : data (operands) are stored in memory

1) concerns around storage architecture

- ↳ where and how do we store operands for computation?
- ↳ where and how do we store the computation results?
- ↳ how do we specify the operands?

2) common storage architecture design

stack

↳ operands are loaded / computed onto a stack, and operations interact w/ that stack

Accumulator

↳ operands are loaded / computed and values are added to those in accumulator

↳ operations interact w/ accumulated value

memory-memory

↳ all operands are directly stored and interacted in memory

general-purpose register

↳ registers : special memory units on CPU

- register-memory
  - ↳ one operand load/stored on register, one directly in memory
- register-register (load-store)

3) GPR architecture for modern processors

↳ GPR most common choice today

↳ RISC typically use load-store

↳ CISC typically use mix of register-register and register-memory

③ memory address and content

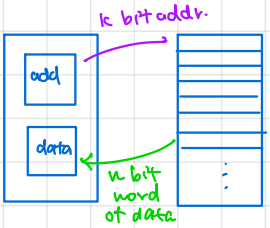
1) memory

↳ given a k-bit address, address space is of size 2^k since you can represent up to 2^k unique values

↳ memory represented as a single-dimensional array that is indexed

↳ each memory transfer consists of one word of n bits

↳ often, $n = k$



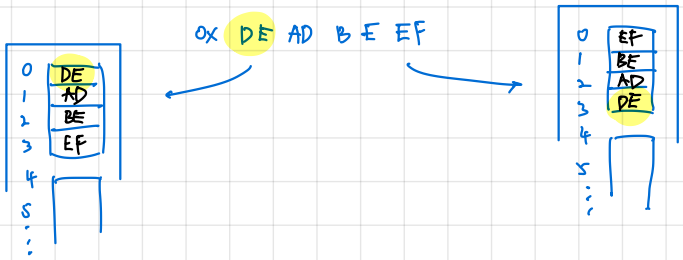
2) memory content: endianness → the relative ordering of bytes in multi byte word stored in memory

big endian

↳ MSByte stored in lowest address

little endian

↳ LSByte stored in lowest address



3) memory addressing modes: how to encode address of operand (in memory, in register, as imm, etc.)

Addressing mode	Example	Meaning	
Register	Add R4,R3	R4 ← R4+R3	⇒ operands stored in registers
Immediate	Add R4,#3	R4 ← R4+3	⇒ operand stored as immediate
Displacement	Add R4,100(R1)	R4 ← R4+Mem[100+R1]	⇒ operand stored in memory, referenced by immediate offset from address in register
Register indirect	Add R4,(R1)	R4 ← R4+Mem[R1]	⇒ same, but offset = 0
Indexed / Base	Add R3,(R1+R2)	R3 ← R3+Mem[R1+R2]	⇒ same, but address is sum of data stored in two registers
Direct or absolute	Add R1,(1001)	R1 ← R1+Mem[1001]	⇒ direct addressing
Memory indirect	Add R1,@(R3)	R1 ← R1+Mem[Mem[R3]]	⇒ address stored in memory, register stores that address. Double retrieval!
Auto-increment	Add R1,(R2)+	R1 ← R1+Mem[R2]; R2 ← R2+d	⇒ register indirect, but register increments after
Auto-decrement	Add R1,–(R2)	R2 ← R2-d; R1 ← R1+Mem[R2]	⇒ register indirect, but decrement first
Scaled	Add R1,100(R2)[R3]	R1 ← R1+Mem[100+R2+R3*d]	⇒ Displacement, but offset by some product immediate value of register value and other constant

④ operations in instruction set

Standard Operations

Data Movement	load (from memory) store (to memory) memory-to-memory move register-to-register move input (from I/O device) output (to I/O device) push, pop (to/from stack)
Arithmetic	integer (binary + decimal) or FP add, subtract, multiply, divide
Shift	shift left/right, rotate left/right
Logical	not, and, or, set, clear
Control flow	Jump (unconditional), Branch (conditional)
Subroutine Linkage	call, return
Interrupt	trap, return
Synchronization	test & set (atomic r-m-w)
String	search, move, compare
Graphics	pixel and vertex operations, compression/decompression

Rank	Integer Instructions	Average %
1	Load	22%
2	Conditional Branch	20%
3	Compare	16%
4	Store	12%
5	Add	8%
6	Bitwise AND	6%
7	Sub	5%
8	Move register to register	4%
9	Procedure call	1%
10	Return	1%
	Total	96%

Make these instructions fast!
Amdahl's law – make the common cases fast!

⑤ instruction formats and encoding

1) instruction length

variable length instructions

↳ instructions vary in byte length
↳ require multi-step fetch and decode
↳ allow for a more flexible but complex and compact instruction set

fixed length instructions

↳ used in most RISC processors
↳ fixed instruction byte length
↳ allow for easy fetch and decode
↳ simplify pipelining and parallelism
⇒ but, instruction bits are scarce

hybrid

↳ mix; some instructions fixed, some variable length

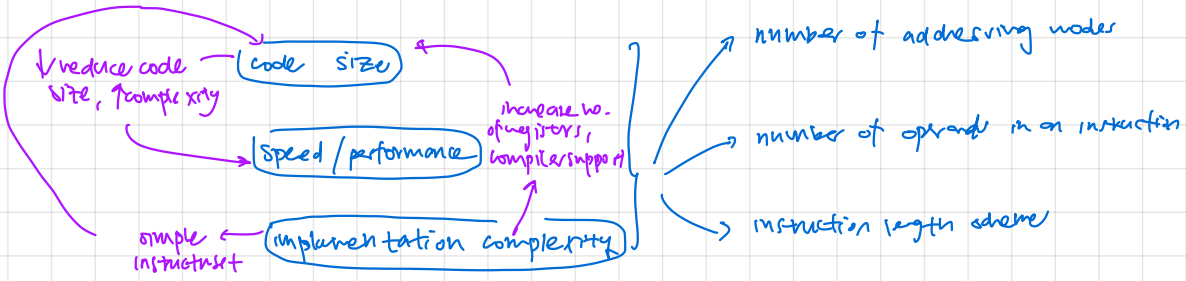
2) instruction fields

↳ instruction must have
- opcode: to specify operation
- operands: values to compute

→ decides size of and how many needed for operation

↳ expected for architecture to have support for 8, 16, 32, 64-bit floating point and integer operation

3) instruction encoding considerations

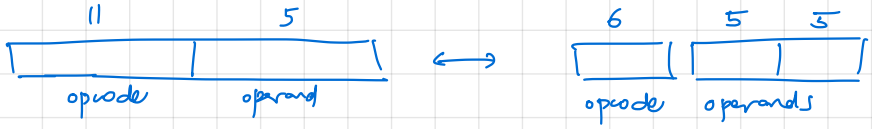


4) fixed length instructions and expanding opcode scheme

↳ need to fit multiple instruction types (w diff operand lengths) into the same no. of bits

↳ use expanding opcode scheme : opcode takes up operand space

eg.



↳ maximising and minimising no. of instructions

- more bits = more instructions 2^k
- use single unique opcodes to distinguish between types
- "stage-wise" splitting

maximise : all but max length = 1

minimise : all but shortest = 1 (if possible)

eg.

- Design an expanding opcode for the following to be encoded in a 36-bit instruction format. An address takes up 15 bits and a register number 3 bits.
  - 7 instructions with two addresses and one register number.
  - 500 instructions with one address and one register number.
  - 50 instructions with no address or register.

One possible answer:

3 bits	15 bits	15 bits	3 bits
000 → 110 opcode	address	address	register
111	000000 + 9 bits opcode	address	register
111	000001 : 110010 + 9 0s opcode	unused	unused

maximise

Start from max, minus off set aside (relative)

eg. opcode sizes : 32 29 16 13 10 (5)

$2^{32} - 2^3 - 2^{16} - 2^{19} - 2^{22} + 5$

$2^3 - 2^{16} - 2^{19} - 2^{22}$: set aside

$+5$: one each

$(2^{29} - 1 - 1\times2^3 - 1\times2^{13} - 1\times2^{16} - 1\times2^{19}) \times 2^3$

↑ typeB ↑ E ↑ C ↑ D ↑ F

minimise

Start from min, cascade down by sharing opcodes

eg. $(2^4 - 1) + (2^4 - 1) + 2^4$

$= 46$

OR

$(2^{10} - 5) + 5$

↑ smallest ↑ all others

intuition : (opcode (of next opcode size) for each x remaining bits

32 max 29

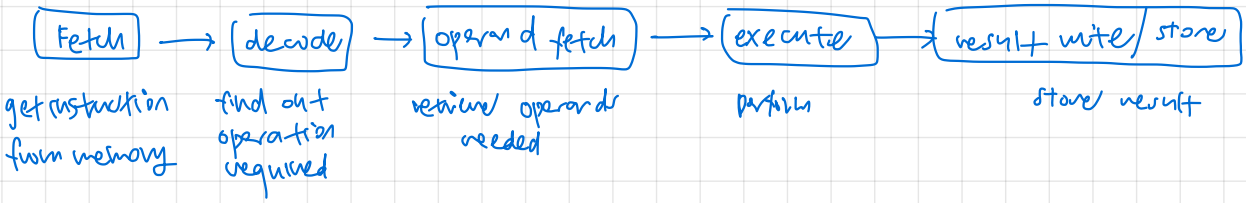
1 29bit opcode x 2^3 unusable

$2^{32} - 1\times2^3 \ldots$

5. Processors : Datapath

collection of components that move data and perform arithmetic, logical and memory operations

① instruction execution cycles



MIPS instruction execution cycle

		`add $3, $1, $2`	`lw $3, 20($1)`	`beq $1, $2, ofst`
①	Fetch	*standard*	*standard*	*standard*
②	Decode			
③	Operand Fetch	○ Read [**$1**] as *opr1* ○ Read [**$2**] as *opr2*	○ Read [**$1**] as *opr1* ○ Use ***20*** as *opr2*	○ Read [**$1**] as *opr1* ○ Read [**$2**] as *opr2*
④	Execute	*Result* = *opr1* + *opr2*	○ *MemAddr* = *opr1* + *opr2* ○ Use *MemAddr* to read from memory	*Taken* = (*opr1* == *opr2*)? *Target* = (**PC**+4) + **ofst**×4
⑤	Result Write	*Result* stored in **$3**	*Memory* data stored in **$3**	if (*Taken*) **PC** = *Target*

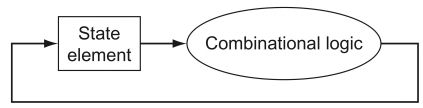
- **opr** = operand
- **MemAddr** = Memory Address
- **ofst** = offset

② The idea of clocking

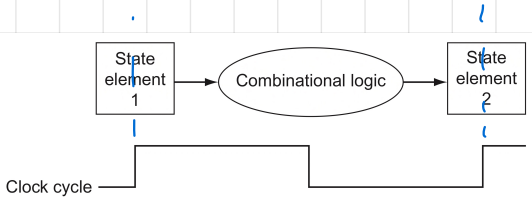
↳ clocking methodology defines when signal can be read and written, to prevent conflict between values

↳ edge-triggered clocking methodology : any values stored in a sequential logic element are updated only on a clock edge (we choose rising edge)

⇒ inputs are values that were written in a previous clock cycle, outputs are values that can be used in next cycle

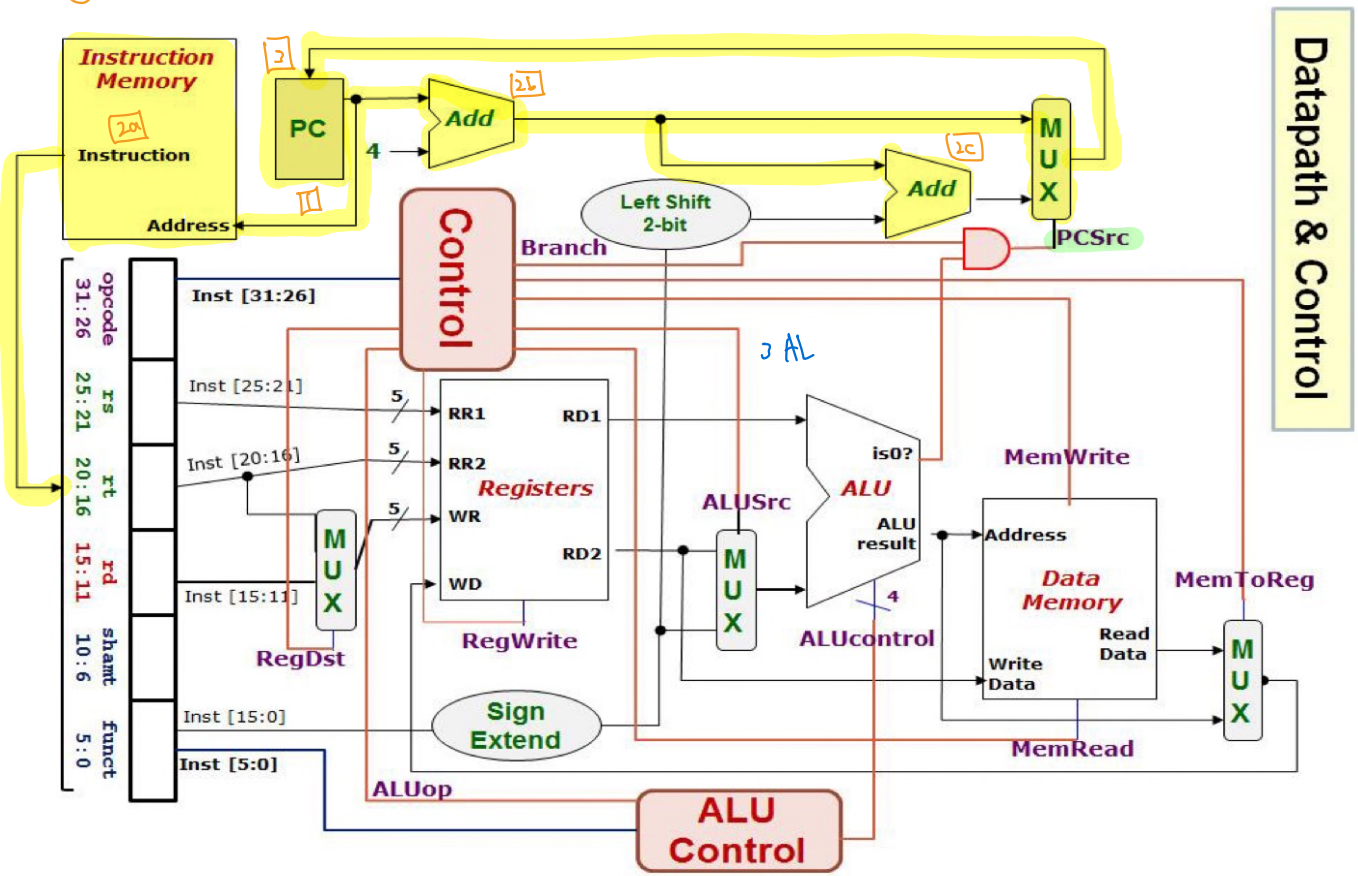


normal: risks conflict or intermediate values



values only updated on rising clock edge → no conflict in execution of logic

① instruction fetch stage



① Instruction Fetch

process

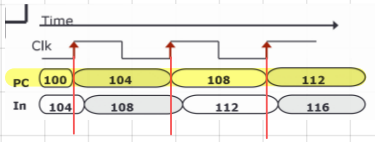
1) During first half of clock period PC is read, sent to memory

2a) instruction is loaded from instruction memory and sent to decode stage

2b) adder adds PC value. PC+4 is also added to sign extended 16 bit immediate.

2c) multiplexer chooses which (PC+4 or PC+4+immediate×4) to return depending on PCSrc

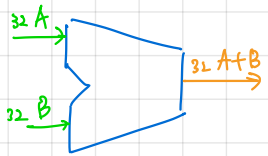
3) PC is updated with new value at the next rising edge



components

1) Adder

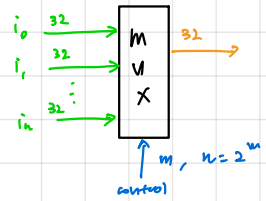
↳ combinational logic circuit to implement addition of two numbers



2) multiplexer

↳ circuit that selects one input from multiple based on m bit control signal

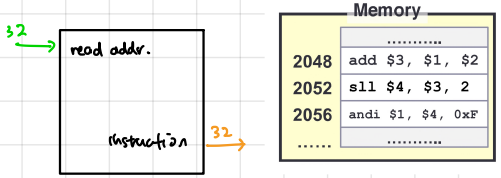
↳ select i_n line if control signal = i



3) instruction memory

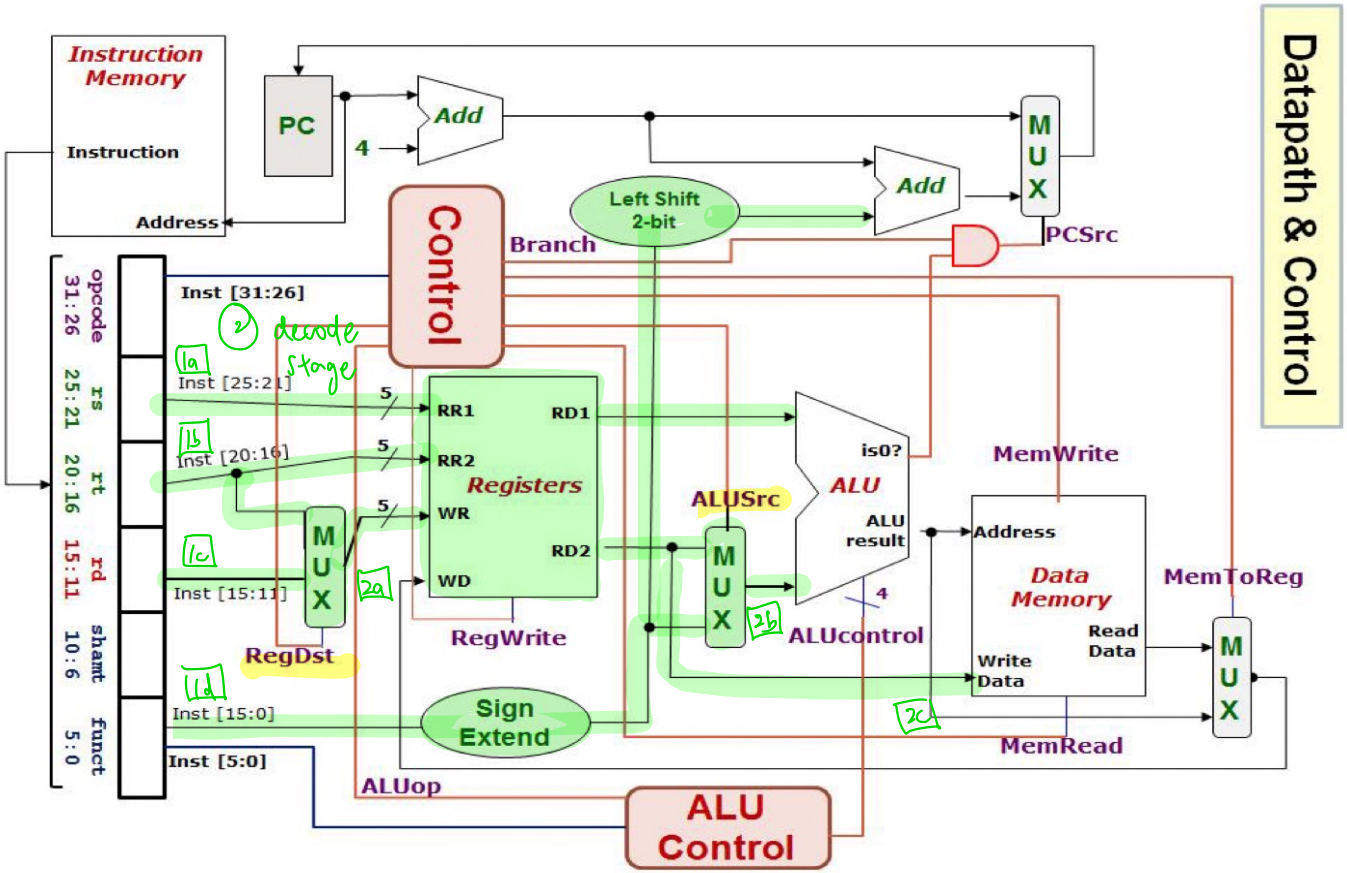
↳ sequential circuit that has internal state to store information

↳ given instruction address as input, outputs content (instruction) at address



4) PC (program counter)

↳ special register that stores address of current instruction being executed



2) Decode stage

Process

1a. value in rs is loaded to RR1 to RD1 and output

1b. value in rt is loaded to RR2, MUX and RD2 and output

1c. value in rd is loaded to MUX1

1d. 16 bit immediate is sign extended and loaded to MUX2, as well as left shifted 2 bits and loaded to adder at fetch stage

2a. MUX1 decides between rd and rt to load into WR using RegDst signal. 0 ⇒ rt, 1 ⇒ rd

2b. MUX2 decides between RD2 and immediate value to send as output using ALUSrc signal.

↳ 0 ⇒ RD2, 1 ⇒ immediate

R mode: op | rs | rt | rd | shift

I mode: op | rs | rt | immed

op: read

2c. RD2 signal is passed to Data memory, for writing to memory if needed. (SW)

Component

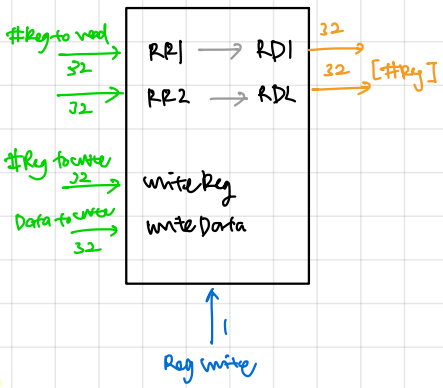
1) register file

↳ a collection of 32 registers

↳ each register is 32 bits wide, can be read/written by specifying reg. number

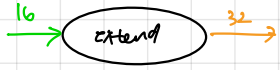
↳ read at most two registers per instruction, write at most one register per instruction

↳ RegWrite is a control signal to indicate writing of register : (True, 0 False)



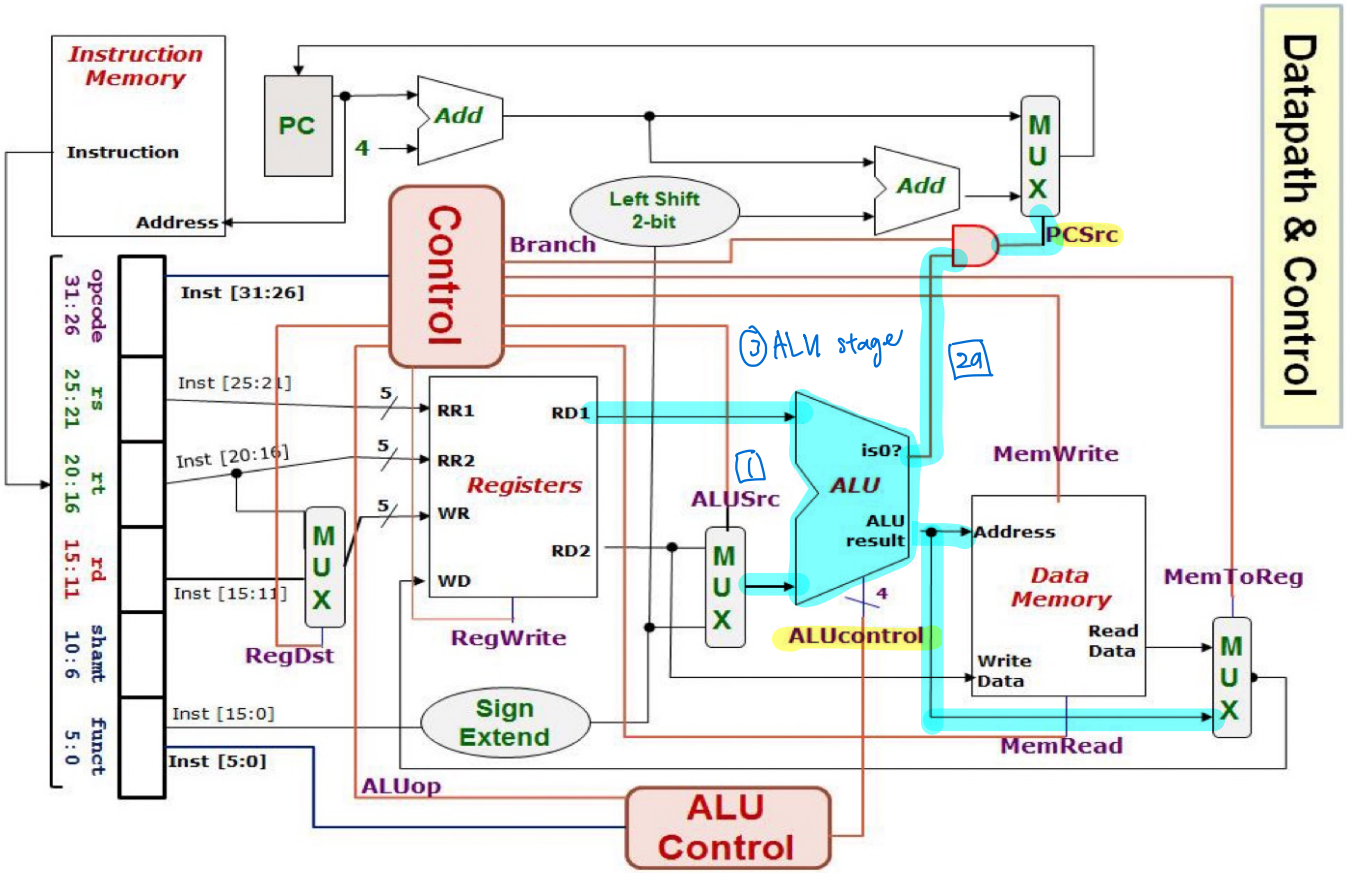
2) sign extend

↳ circuit to extend 16 bit signal to 32 bits



3) 2-bit left shift

↳ circuit to shift 32 bit signal



③ ALU computation

Process

1. operands are read into ALU

2a. ALU does op1 - op2 to test if op1 == op2. signal is sent to AND gate to form control signal PCSrc ← control output, to decide if next instruction is PC+4 or PC+4 + immediate x4

2b. ALU does op1 + op2, sends it to memory as well as MUX for later decision of whether to send read data from memory or result directly from ALU.

Components

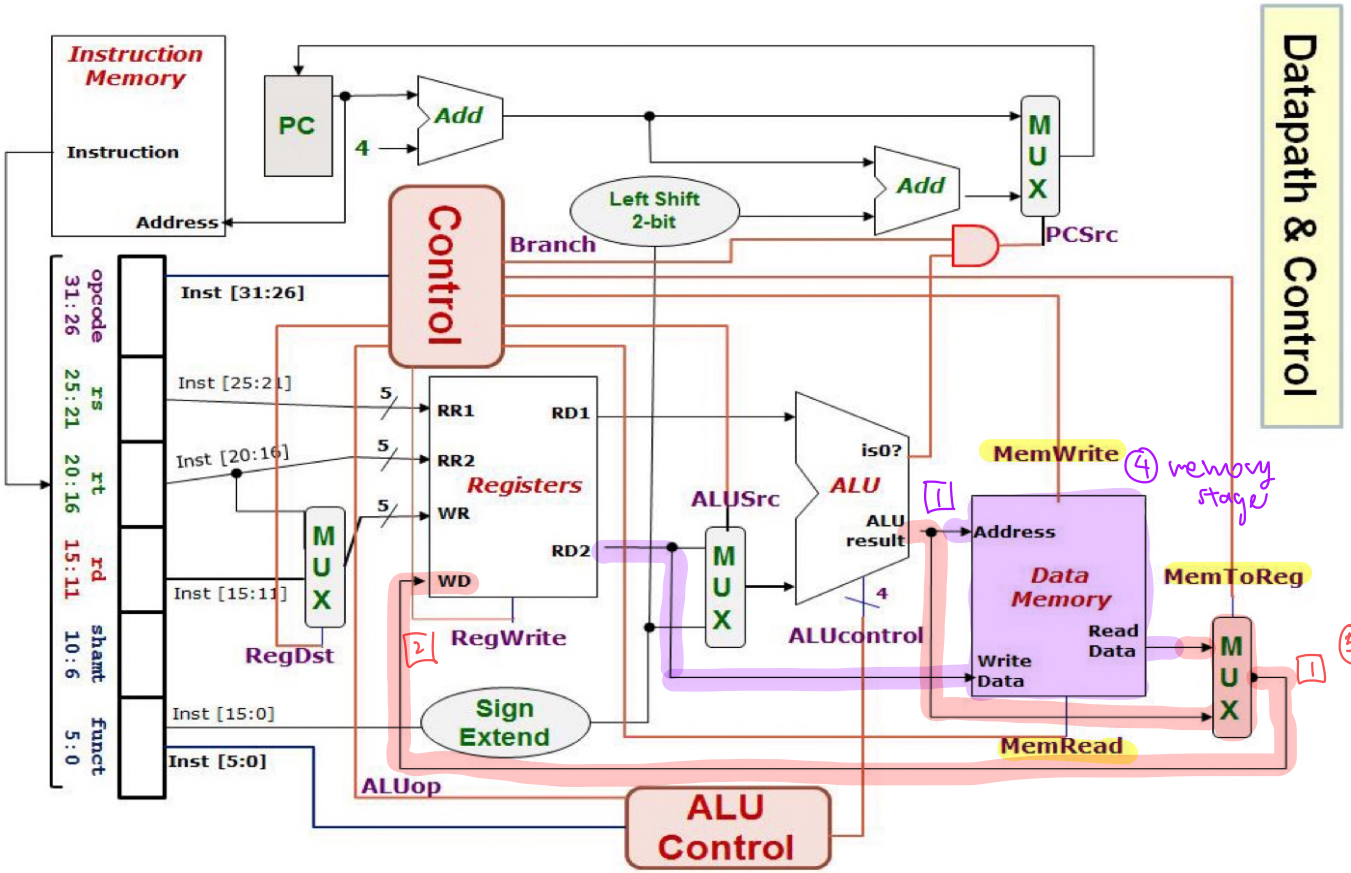
1) Arithmetic logic unit

↳ combinational circuit to implement arithmetic and logic operations

↳ uses 4 bit ALUctl control signal to indicate operation

↳ outputs 32 bit result and 1 bit op1 == op2 result

ALUcontrol	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR



4 memory stage

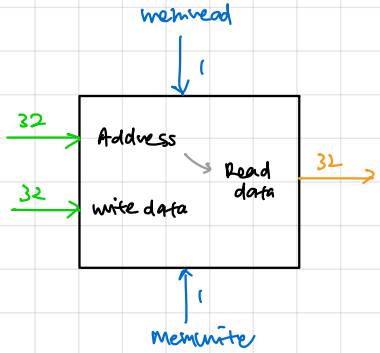
1. output from ALU is read as an address and output to RD to MUX.
2. if memWrite = 1, then output from RD2 is written to memory at address from ALU.

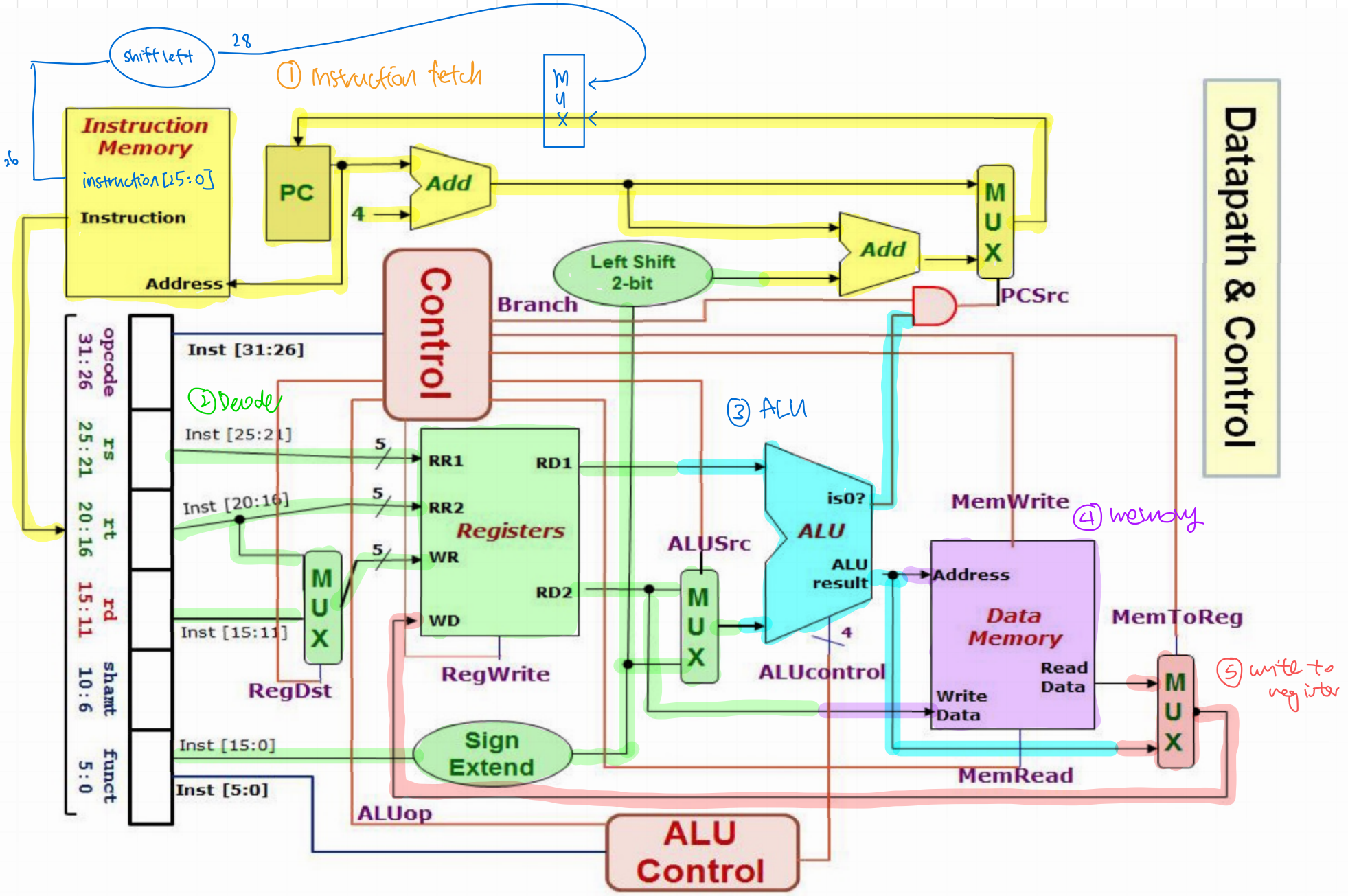
5 write stage

1. MUX routes correct result to register file depending on MemToReg signal
2. written in register WR determined at decode stage

components

1) Data memory
   - storage element for data
   - takes in memory address from ALU and data to be written from RD2
   - depending on memWrite / memRead signals, will write to memory or output to read data. only 1 can be asserted at a time. so signal is default 1 0000 ...)



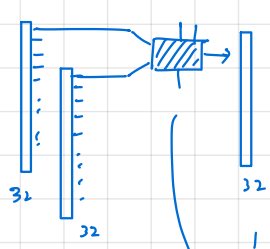

Datapath & Control
shift left
28
26
① Instruction fetch
M U X
Instruction Memory
instruction[25:0]
Instruction
Address
PC
4
Add
Add
M U X
Left Shift 2-bit
Branch
PCSrc
Control
Inst [31:26]
② Decode
③ ALU
Inst [25:21]
Inst [20:16]
Inst [15:11]
Inst [15:0]
Inst [5:0]
5
5
5
RR1
RR2
WR
WD
RD1
RD2
Registers
M U X
RegDst
RegWrite
ALUSrc
M U X
is0?
ALU
ALU result
4
ALUcontrol
MemWrite
④ memory
Address
Data Memory
Read Data
Write Data
MemRead
MemToReg
M U X
⑤ write to register
Sign Extend
ALUop
ALU Control
opcode 31:26
rs 25:21
rt 20:16
rd 15:11
shamt 10:6
funct 5:0

Notes

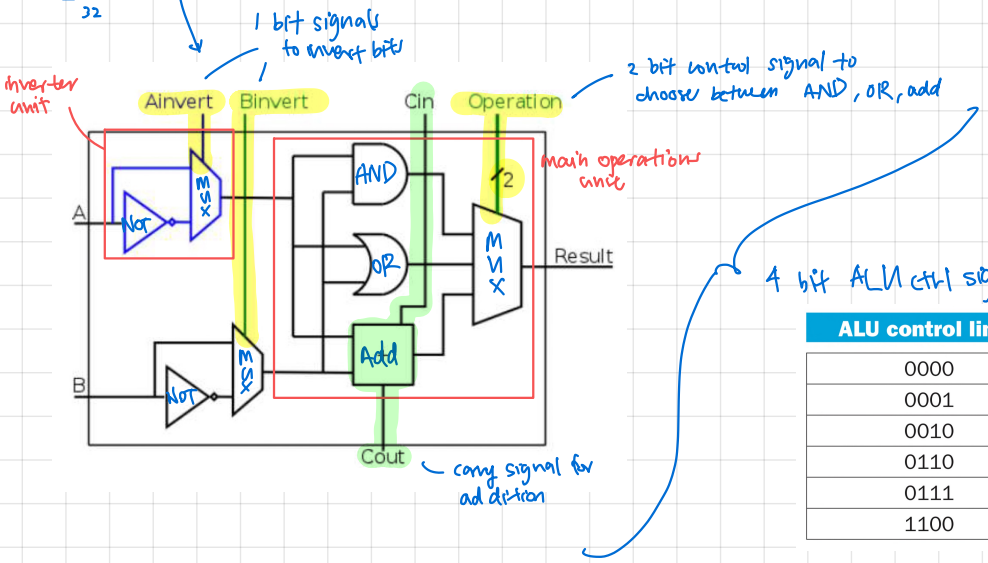
1. Things happen in parallel. Look for multiple paths that are sequential → critical path = the one with the longest time required
(be esp. careful @ MUX)

6. Control using combinational circuits to generate desired control signal based on opcode and function code

① how an ALU works



each bit is wired to an ALU sub unit



4 bit ALU ctrl signal needed

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

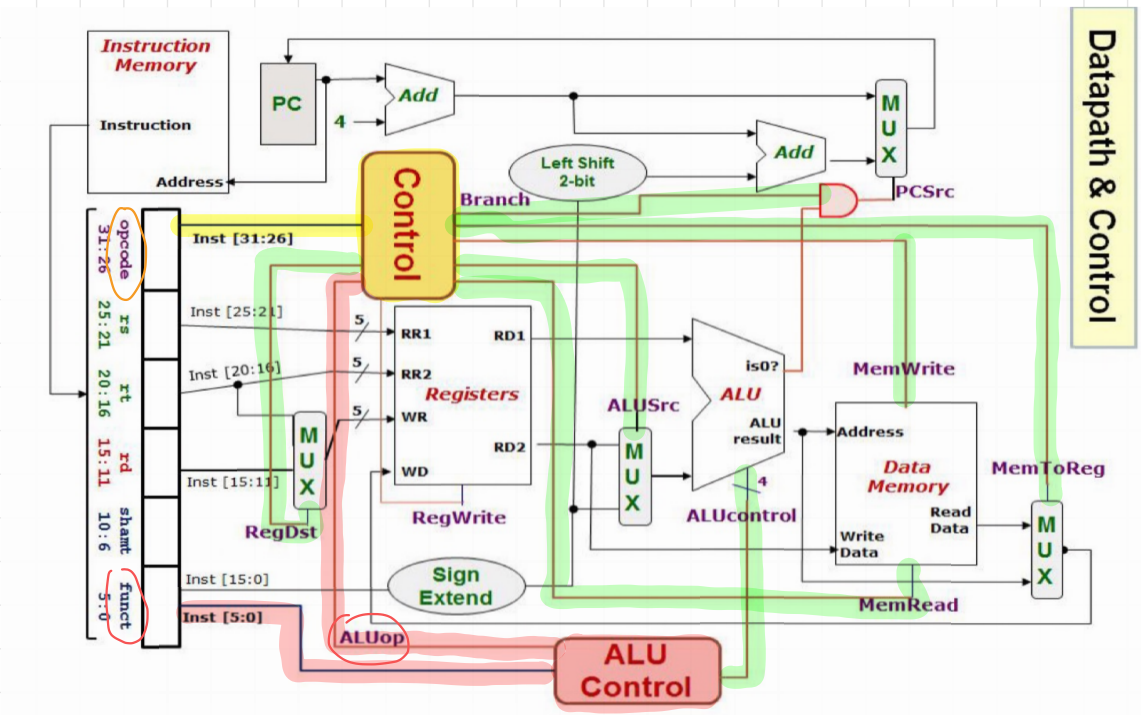
② encoding and decoding

↳ opcode + func gives 11 bits ⇒ but don't need that many. Doing brute force way = inefficiencies due to implementation complexity

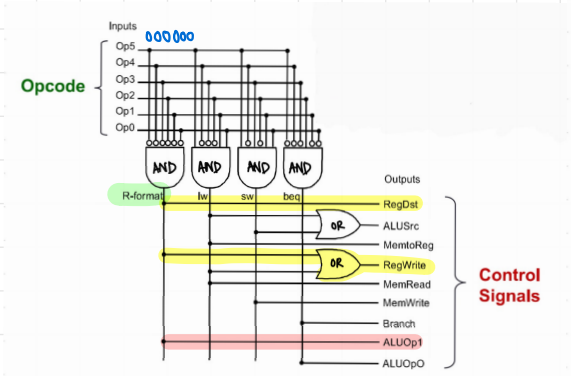
6 bit opcode → encode 6 fields → generate 2 bit ALUop (code)

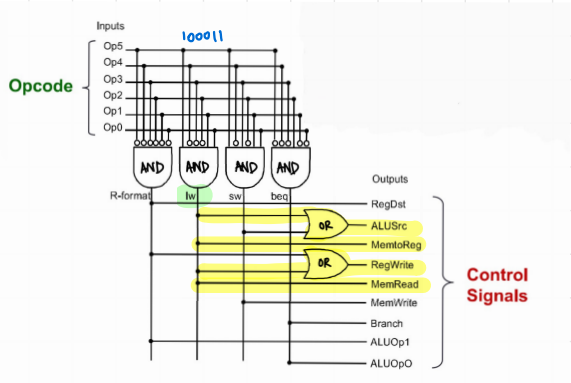
5 bit func code → 4 bit ALUsrc

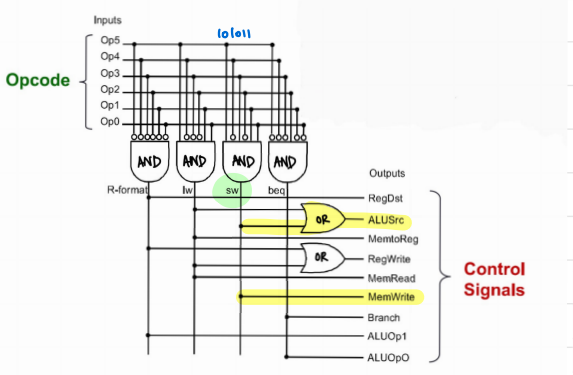
design arrangement based on specific combination of signals you need; no strictly about "unique instruction identifier". Use truth table to see combination needed → decide op code and logic gates needed to switch on those combinations of signals for each instruction

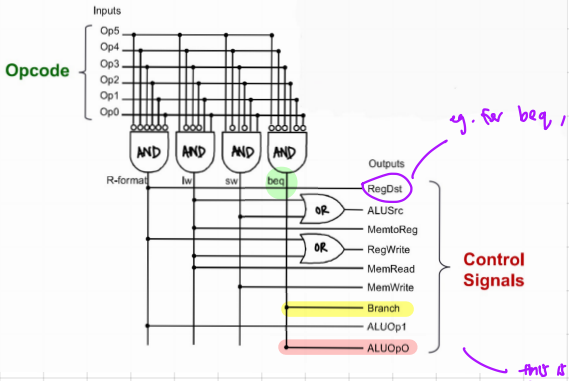


1) generating control signals w combinational circuit (control)









eg. for beq, x: doesn't matter

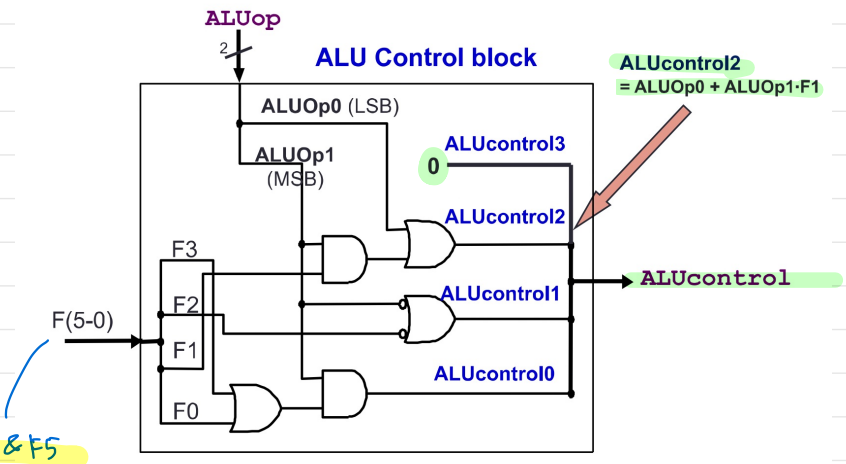
(rt, depends on implementation of how digital circuit is simplified

this is just one implementation

	RegDst	ALUSrc	MemTo Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop	
								op1	op0
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

	Opcode (Op[5:0] == Inst[31:26])						
	Op5	Op4	Op3	Op2	Op1	Op0	Value in Hexadecimal
R-type	0	0	0	0	0	0	0
lw	1	0	0	0	1	1	23
sw	1	0	1	0	1	1	2B
beq	0	0	0	1	0	0	4

2) generating ALU ctrl using combinational circuit



F4 & F5 are not used at all for the standard instruction set (simplified diagram)

	ALUop		Funct Field (F[5:0] == Inst[5:0])						ALU control
	MSB	LSB	F5	F4	F3	F2	F1	F0	
lw	0	0	X	X	X	X	X	X	0 0 1 0
sw	0	0	X	X	X	X	X	X	0 0 1 0
beq	0 X	1	X	X	X	X	X	X	0 1 1 0
add	1	0 X	1 X	0 X	0	0	0	0	0 0 1 0
sub	1	0 X	1 X	0 X	0	0	1	0	0 1 1 0
and	1	0 X	1 X	0 X	0	1	0	0	0 0 0 0
or	1	0 X	1 X	0 X	0	1	0	1	0 0 0 1
slt	1	0 X	1 X	0 X	1	0	1	0	0 1 1 1

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

③ cycle implementation

1) cycle time : clock period = assumption of worst case ⇒ longest time

↓

single cycle implementation means losses when faster operations are being run

2) multicycle implementation : each execution step is one clock cycle, instructions take variable no. of clock cycles to complete. cycle time = time of longest execution step

3) pipelining

7. Boolean algebra

① Boolean values and operators

1) operator precedence NOT (') > AND (·) > OR (+)

2) laws & theorems

Identity laws	Dual form
A + 0 = 0 + A = A	A · 1 = 1 · A = A
Inverse/complement laws	
A + A' = 1	A · A' = 0
Commutative laws	
A + B = B + A	A · B = B · A
Associative laws *	
A + (B + C) = (A + B) + C	A · (B · C) = (A · B) · C
Distributive laws	
A · (B + C) = (A · B) + (A · C)	A + (B · C) = (A + B) · (A + C)

Idempotency	Dual form
X + X = X	X · X = X
One element / Zero element	
X + 1 = 1	X · 0 = 0
Involution	
(X')' = X	
Absorption 1	
X + X·Y = X	X·(X + Y) = X
Absorption 2	
X + X'·Y = X + Y	X·(X' + Y) = X·Y
DeMorgans' (can be generalised to more than 2 variables)	
(X + Y)' = X' · Y'	(X · Y)' = X' + Y'
Consensus	
X·Y + X'·Z + Y·Z = X·Y + X'·Z	(X+Y)·(X'+Z)·(Y+Z) = (X+Y)·(X'+Z)

3) duality: if the AND/OR operators & 1/0 in a boolean equation are swapped, the equation remains valid.
↳ logically equivalent. ⇒ useful in proofs or simplification

4) Boolean functions: Logic equations, taking input variables to produce an output boolean value

② Standard forms

1) building blocks

literal: single boolean variable
x x'

product term: logical product of literals or a single literal
x·y·z', x

sum term: logical sum of literals or a single literal
x, x+y'+z

2) combinations

sum of products: a single product term or a logical sum of several
x x + y x + y·z y·z + x·y'

product of sums: a single sum term or logical product of several sum terms
x x·(y+z) (x+y)·(x+z)

③ Canonical standard forms

1) minterms & maxterms of n variables → contain n literals from all the variables

product term ↔ sum term (complements)

x	y	Minterms		Maxterms	
		Term	Notation	Term	Notation
0	0	x'·y'	m0	x+y	M0
0	1	x'·y	m1	x+y'	M1
1	0	x·y'	m2	x'+y	M2
1	1	x·y	m3	x'+y'	M3

notice its inversion when 0,0

complements

2) canonical SOP and POS
↳ SOP of minterms / POS of max terms

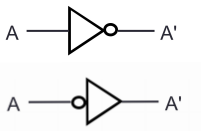
3) conversion between SOP and POS ⇒ complement

$\Sigma(1,4,5,6,7) = \Pi M(0,2,3)$

8. Logic circuits

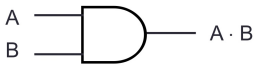
① Logic gates and truth tables

NOT



A	A'
0	1
1	0

AND

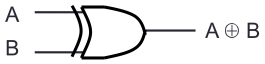


A	B	A · B
0	0	0
0	1	0
1	0	0
1	1	1

OR

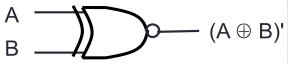
A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	1

XOR



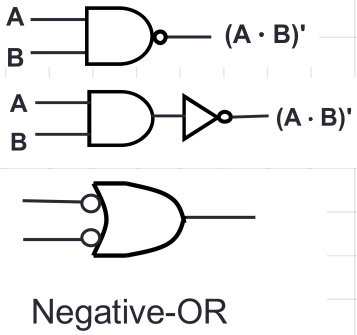
A	B	A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

XNOR



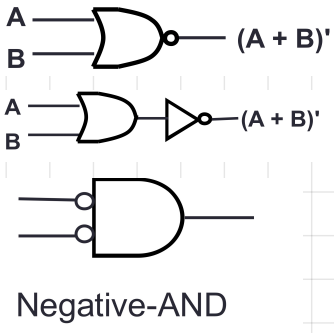
A	B	(A ⊕ B)'
0	0	1
0	1	0
1	0	0
1	1	1

NAND



A	B	(A · B)'
0	0	1
0	1	1
1	0	1
1	1	0

NOR



A	B	(A + B)'
0	0	1
0	1	0
1	0	0
1	1	0

② Logic circuits

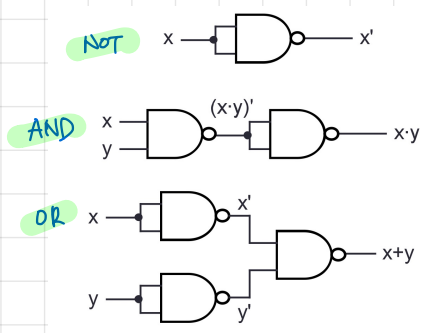
1) Gate fan in & fan out
   - fan in: no. of inputs
   - fan out: no. of gates a gate can deliver its output to

2) circuit analysis

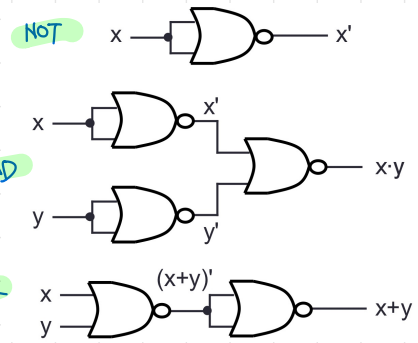
③ Complete logic sets and universal gates

1) complete sets of logic (AND / OR / NOT) with NAND & NOR, also (AND, NOT)

NAND

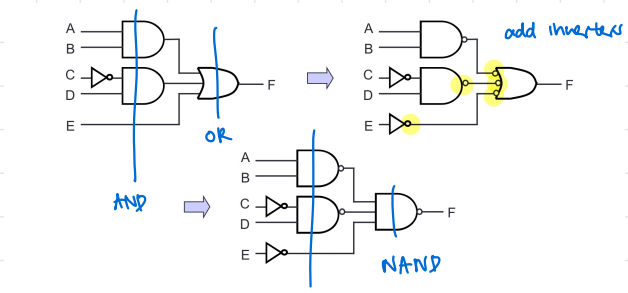


NOR

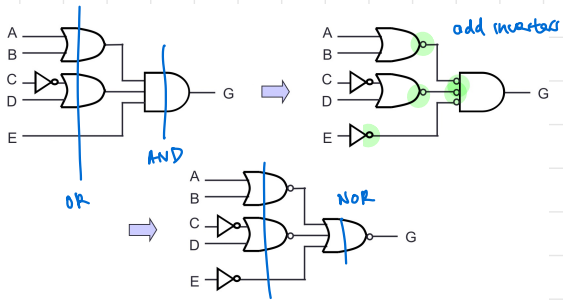


2) 2-level circuits (NOT discounted)

SOP AND/OR | NAND

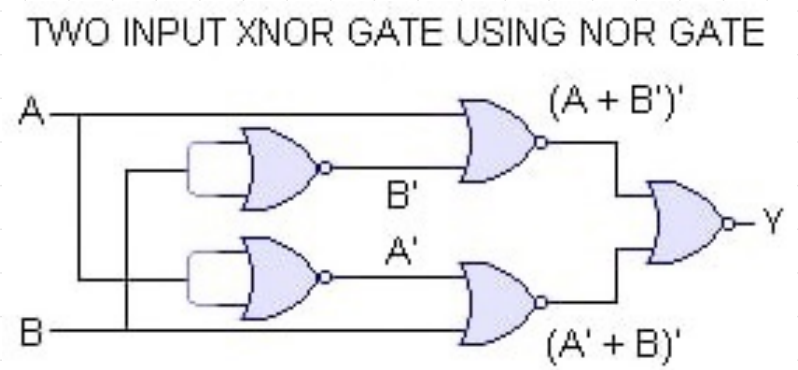


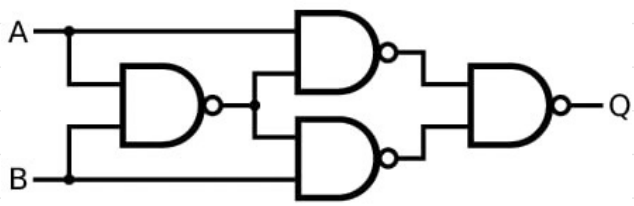
POS OR/AND | NOR



Notes :

1. Common gates





9. Boolean function simplification

① algebraic simplification

↳ using laws & theorems to simplify given expression

↳ absorbtion, distributive, consensus most useful in reducing terms

② K maps

↳ systematic method to obtain simplied SOP expression in fewest product terms & literals, by visualising minitems on matrix, grouping them & simplifying based on complement law

↳ $A + A' = 1$ · $1 \cdot B = B$. In each K-map, a cell containing a '1' corresponds to a miniterm of a given function F where the output is 1. By arranging adjacent cells to only differ by 1 literal and grouping them in powers of two, we can 'factorise out' shared components (use distributive law) and simplify using complements thus reducing no. of terms & literals.

eg. $a \cdot b \cdot c \cdot d' + a \cdot b \cdot c \cdot d$ (adjacent terms)

$= (d' \cdot d)(a \cdot b \cdot c)$

1) layout, rules & groupings

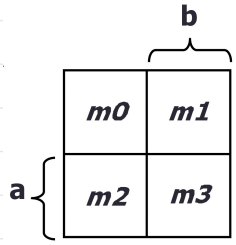
↳ each square represents a miniterm

↳ adjacent squares represent minitems that differ by exactly 1 literal

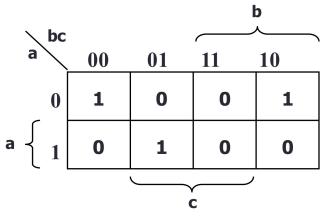
↳ a wrap around exists, so every cell has n neighbours. n variable map ⇒ 2^n cells.

↳ group terms in powers of 2 : 1, 2, 4, 8, 16, 32

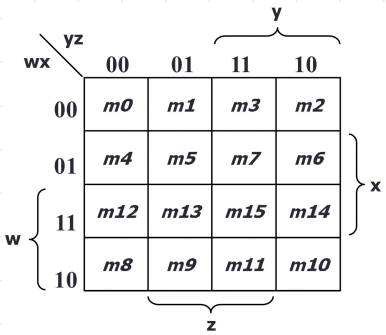
2 variable



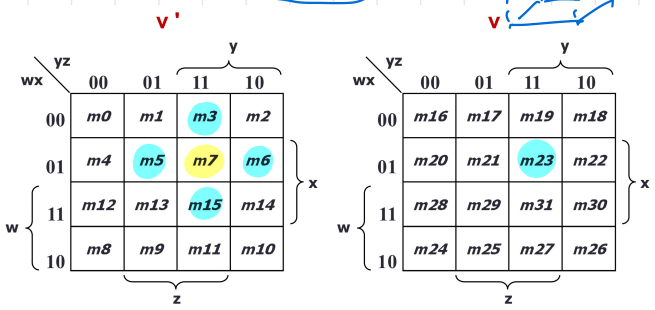
3 variable



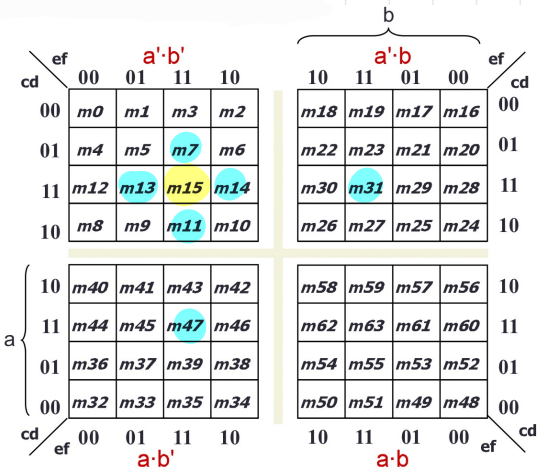
4 variable



5 variable stacks



6 variable mirror



2) PIs and EPIs

↳ prime implicant that includes a term not covered by any other PI

biggest grouping possible involving those terms

(implicant : product term that could cover minitems in a function)

3) Don't care terms

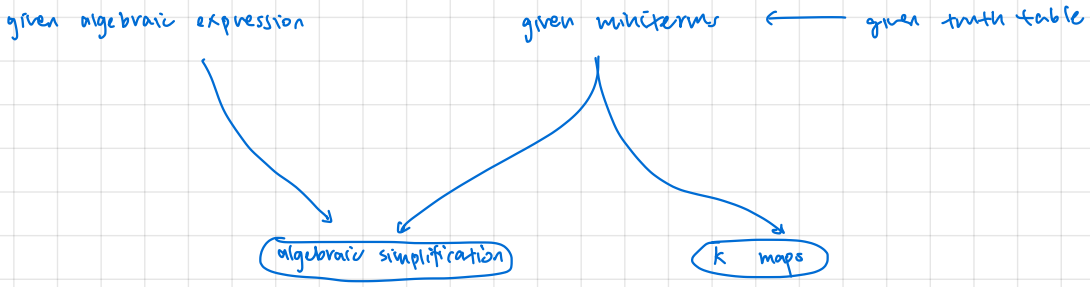
↳ can substitute for 1 or 0, depending on needs

3) Finding SOP expressions

1. create cell map s.t. terms will differ by 1 literal.
2. populate the K map. 1 → miniterms, 0 → maxterms
3. circle all largest possible groups in powers of two (PIs) – don't forget wrap arounds
4. identify all EPIs (any PIs that have a non-overlapped cell)
5. select EPIs. select minimum no. of PIs to cover all remaining cells.

4) Finding POS expressions

↳ group based on 0s. Equivalent to grouping 1s in K map of F'.
↳ obtain SOP for F'. De morgan's.



10. Combinational circuits

① Combinational circuits

1) each output depends on immediate inputs

⇒ analysis :

1. identify inputs & outputs
2. trace & obtain boolean expression
3. draw truth table

② Design principles

Gate level design

1. define problem : inputs, outputs, truth table
2. simplify function to obtain boolean expressions
3. implement logic circuit

block level design

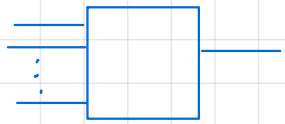
↳ using implemented blocks to implement a function

1. draw truth table
2a. look for patterns that blocks can handle / cascades
2b. look for patterns that blocks not meant to handle, but can handle

③ circuit delays

↳ logic gate w/ delay t ; inputs stable at $t_1, t_2 \ldots$

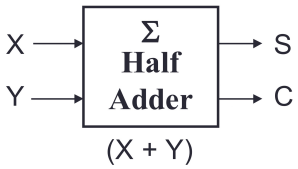
⇒ time at which output is stable is $\max(t_1, t_2 \ldots) + t$

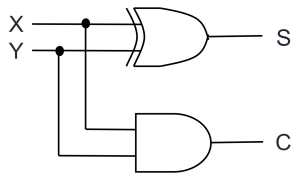


$\max(t_1 \ldots t_i) + t$

---

half adder implements $X + Y \rightarrow$ sum + carry



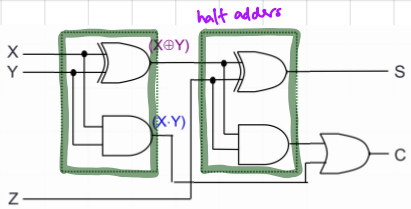


X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$C = X \cdot Y$

$S = X \oplus Y$

full adder implements $X + Y + Z$ (carry) $\rightarrow$ sum + carry



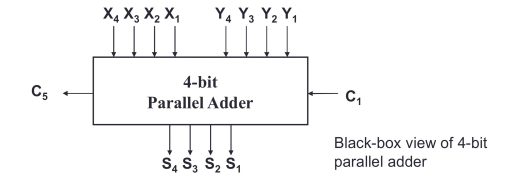
$C = X \cdot Y + (X \oplus Y) \cdot Z$

$S = X \oplus Y \oplus Z$

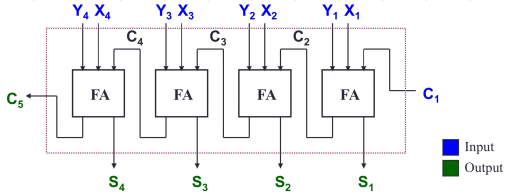
X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

---

parallel adders 9 bit in, 5 bits out. normally implemented by cascading



Black-box view of 4-bit parallel adder



BCD to excess 3 converter converts 0 to 9 to excess 3 representation (BCD + 3)

	BCD				Excess-3			
	A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0
10	1	0	1	0	X	X	X	X
11	1	0	1	1	X	X	X	X
12	1	1	0	0	X	X	X	X
13	1	1	0	1	X	X	X	X
14	1	1	1	0	X	X	X	X
15	1	1	1	1	X	X	X	X

$W = A + B \cdot C + B \cdot D$

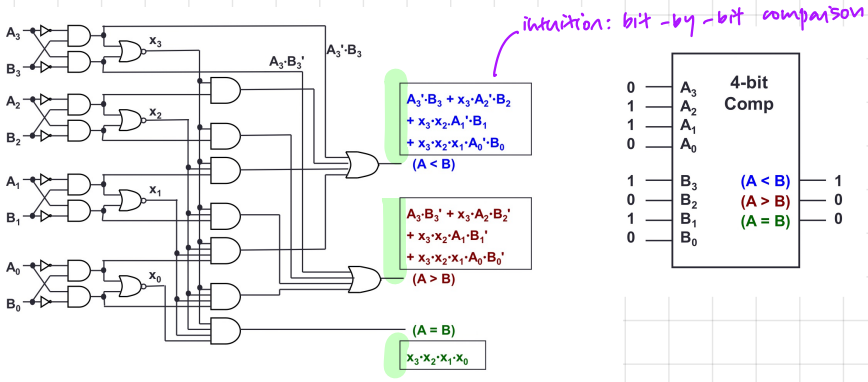
$X = B' \cdot C + B' \cdot D + B \cdot C' \cdot D'$

$Y = C \cdot D + C' \cdot D'$

$Z = D'$

magnitude comparators compares unsigned values to check if $A > B$, $A = B$, $A < B$

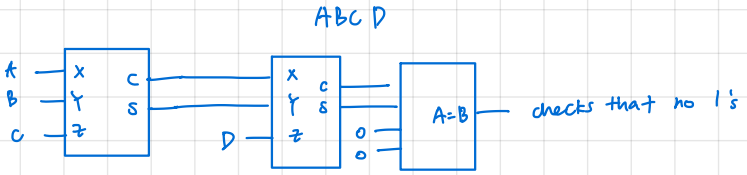
↳ to consider usage, see if one column always one (ie >), or equal, etc.



---

⑤ problem solving: common cases

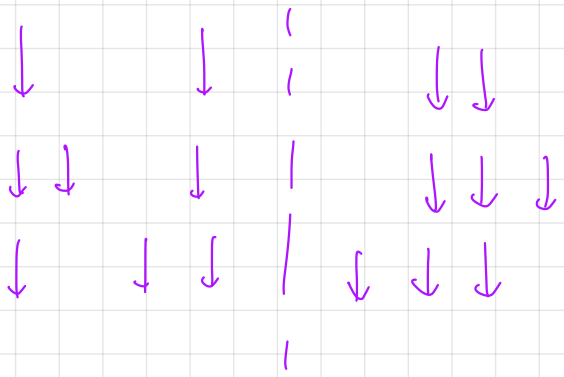
1. to count the number of 1s: add all positions eg. using full adders + magnitude comp.



2. To multiply / divide ⇒ do right or left shifts

3. when examining truth table, pay attention to adjacent output = 1 terms to identify what matters & what doesn't

4. systematically look at subgroups □ □ □ □, commonly first / last n



5. when a lot eg $\Sigma m(1, 2, 4, 5, 6, 7 \ldots)$ ⇒ consider using SOP for F' instead, then use D or NOR gate

11. MSI components

① Integrated circuits

↳ set of electronic circuits on one chip of semiconductor material

↳ 'scale' of integration: number of components ⇒ MSI : 10–500 transistors, 13–99 gates

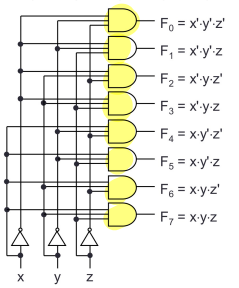
↳ enable control signal : an additional control signal that turns 'off' all outputs or on
⇒ can be '1' enable or '0' enable

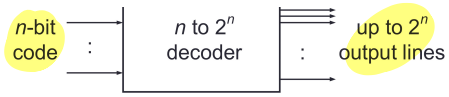
↳ active high / active low outputs

② Decoders

how it works : selects an output line from 2^n of them using a n bit input, $A \cdot B' \cdot C'$ ⇒ AND only 1 on

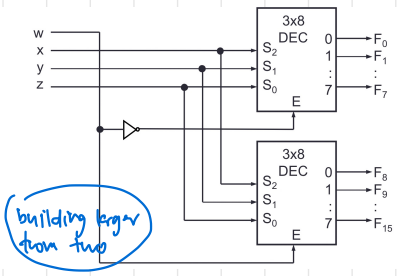
x	y	z	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

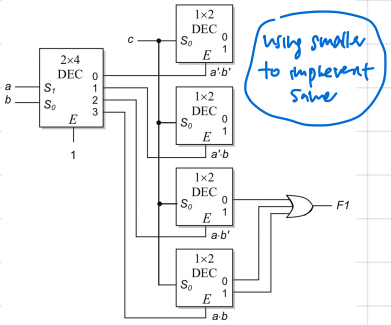




"composite" decoders : use of another bit & gates to choose between decoders and their output ones using enable

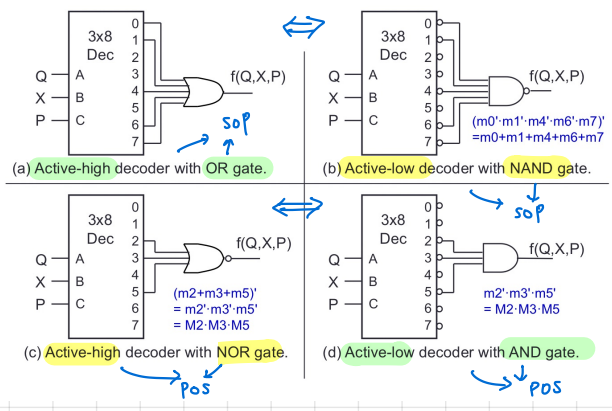
↳ can even use decoder to choose decoder





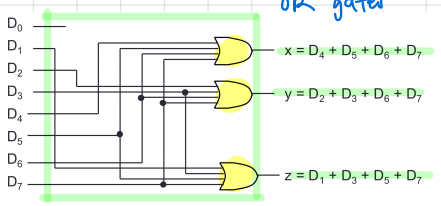
implementing functions boolean function in SOP, use gates

(2/2) $f(Q,X,P) = \Sigma m(0,1,4,6,7) = \prod M(2,3,5)$



③ Encoders

how it works compresses 2^n input to n bits using OR gates



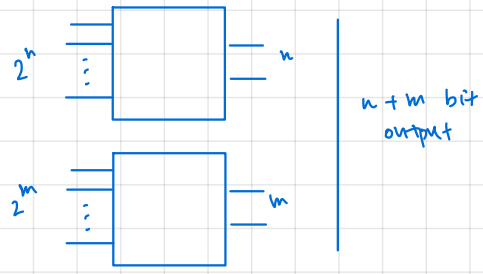
An 8-to-3 encoder

Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

priority encoders if two or more inputs = 1, input w the highest priority takes precedence.

Inputs				Outputs		
D_0	D_1	D_2	D_3	f	g	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

constructing larger encoders



④ Multiplexers

how it works 2^n input lines, n selection lines, m (identical) output lines. "$2^n : m$ mux."

↳ implemented by AND of input & decoder selection, then OR.

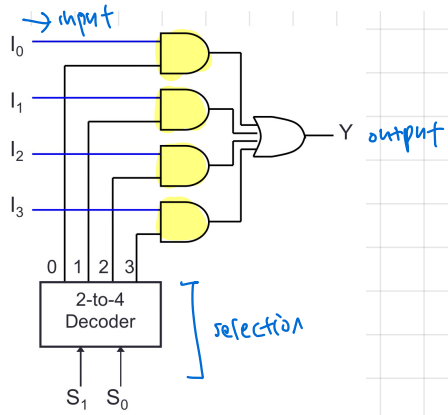
Output of multiplexer is

"sum of the (product of *data lines* and *selection lines*)"

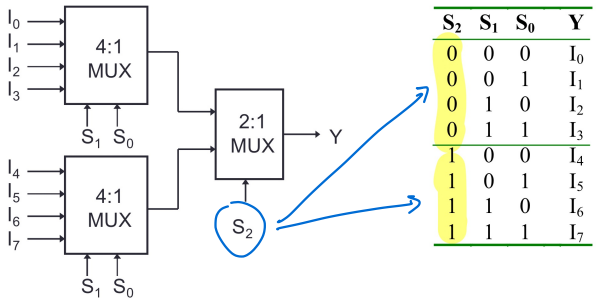
Example: Output of a 4-to-1 multiplexer is:

$$Y = I_0 \cdot (S_1' \cdot S_0') + I_1 \cdot (S_1' \cdot S_0) + I_2 \cdot (S_1 \cdot S_0') + I_3 \cdot (S_1 \cdot S_0)$$

0	1
1	0
1	1



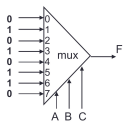
constructing larger multiplexers breaking up bits to choose ⇒ use MSB



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

implementing functions boolean SOP functions

1. Express in sum-of-minterms form.
   Example:
   $F(A,B,C) = A'\cdot B'\cdot C + A'\cdot B\cdot C + A\cdot B'\cdot C + A\cdot B\cdot C'$
   $= \Sigma\, m(1,3,5,6)$
2. Connect n variables to the n selection lines.
3. Put a '1' on a data line if it is a minterm of the function, or '0' otherwise.



This method works because:

Output = $I_0\cdot m_0 + I_1\cdot m_1 + I_2\cdot m_2 + I_3\cdot m_3 + I_4\cdot m_4 + I_5\cdot m_5 + I_6\cdot m_6 + I_7\cdot m_7$

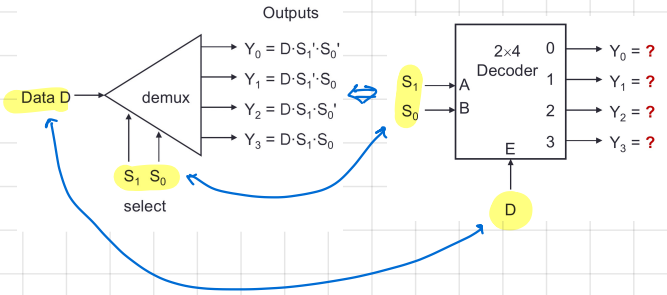
Supplying '1' to I_1, I_3, I_5, I_6, and '0' to the rest:

Output = $m_1 + m_3 + m_5 + m_6$

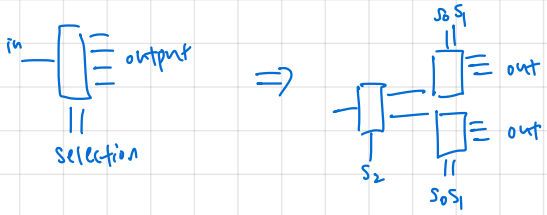
⑤ Demultiplexers

how it works given data input and n bit selection line, directs data (1 bit) to one of 2^n lines.

↳ identical to encoder, where data ≈ enable



constructing comp. demultiplexers

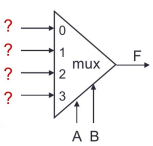


using smaller multiplexers

3. Draw the truth table for function, by grouping inputs by selection line values, then determine multiplexer inputs by comparing input line (C) and function (F) for corresponding selection line values.

A	B	C	F	MUX input
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	

regardless of C, F = 1
F = 0
regardless of C, F = 0
F = C'



Notes:

1. too many minterms → sometimes can implement F' ⇒ use inverter at output
2. can think of enable as an input also

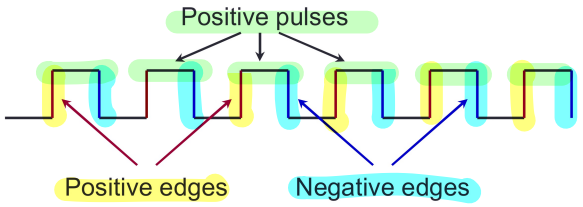
12. Sequential circuits

① memory element

↳ a device that can remember a value indefinitely or change depending on an input

↳ with clock: 'enable' signal of sorts

- pulse triggered
  - ↳ known as latches
  - ON = 1
  - OFF = 0
- edge triggered
  - ↳ known as flip flops
    - positive
      - ON = from 0 to 1
      - OFF = otherwise
    - negative
      - ON = 1 to 0
      - OFF = otherwise



↳ how it works: previous state, enable signal & control signals as inputs, so that Q^+ is also dependent on Q. When off, output does not change regardless of input

② synchronous and asynchronous inputs (for edge triggered units)

↳ synchronous inputs: data on inputs transferred into flip flops output only on triggered edge of clock pulse

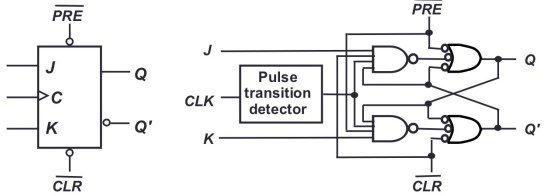
↳ asynchronous input: affect state of the flip flop independent of clock. Think of them as 'master' switches that dictate the state immediately.

↳ can be active low or active high

preset ⇒ Q = 1 immediately

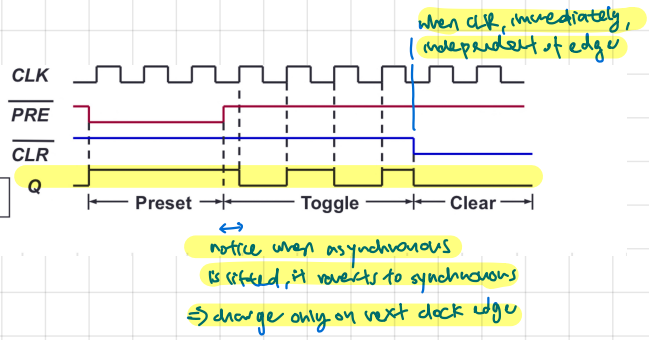
clear ⇒ Q = 0 immediately

eg A J-K flip-flop with active-low PRESET and CLEAR asynchronous inputs.



J = K = HIGH

↳ toggle



③ analysis of sequential circuits

↳ sequential circuits: built from logic gates and flip flops

↳ intuition behind sequential circuit: for outputs that are previous-state dependent eg. vending machine

1. identify and derive boolean expressions for flip flop inputs and system outputs
2. derive state equations for flip flop states
3. derive state table ⇒ m flip flops (states) and n inputs ⇒ 2^{m+n} rows
   - ↳ fill in inputs & states combinations
   - ↳ use flip flop input functions to get inputs eg. KA, JA ...
   - ↳ use characteristic tables & output function to get new states & outputs
4. draw state diagrams

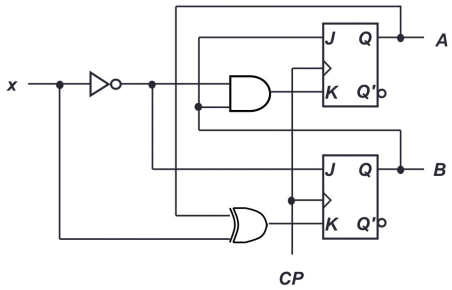


Figure 2

Obtain the flip-flop input functions from the circuit:

$JA = B$ $\quad JB = x'$

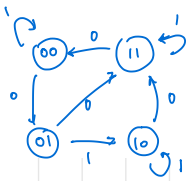
$KA = B \cdot x'$ $\quad KB = A' \cdot x + A \cdot x' = A \oplus x$

$JA = B$ $\quad JB = x'$

$KA = B \cdot x'$ $\quad KB = A' \cdot x + A \cdot x' = A \oplus x$

Fill the state table using the above functions, knowing the characteristics of the flip-flops used.

J	K	Q(t+1)	Comments
0	0	Q(t)	No change
0	1	0	Reset
1	0	1	Set
1	1	Q(t)'	Toggle



Present state A	Present state B	Input x	Next state A^+	Next state B^+	Flip-flop inputs JA	Flip-flop inputs KA	Flip-flop inputs JB	Flip-flop inputs KB
0	0	0	0	0	0	0	1	0
0	0	1	0	1	0	0	0	1
0	1	0	1	0	1	1	1	0
0	1	1	1	1	1	0	0	1
1	0	0	1	0	0	0	1	1
1	0	1	1	1	0	0	0	0
1	1	0	0	0	1	1	1	1
1	1	1	1	1	1	0	0	0

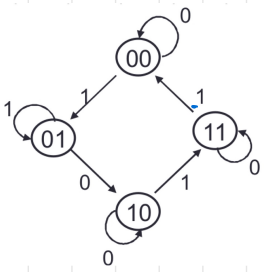
↳ sinks & self connection

↳ state that once a circuit enters, never leaves out

↳ if circuit is able to (without inputs) move from an invalid to valid state in a finite no. of steps

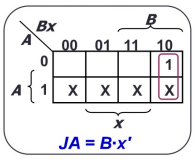
④ Design of sequential circuits

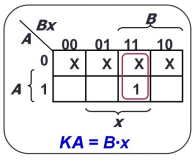
1. From state diagram identify no. of states (flipflops), inputs and outputs
2. Fill in state table
   - ↳ fill in combinations of states & inputs
   - ↳ from state diagram fill in next states
   - ↳ using excitation table, fill in flip flop inputs
   - ↳ use k maps to derive flip flop input expressions, implement

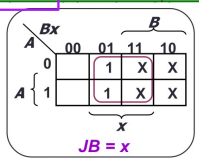


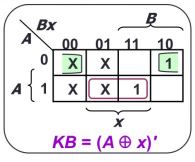
From state table, get flip-flop input functions.

Present state A	Present state B	Input x	Next state A^+	Next state B^+	Flip-flop inputs JA	Flip-flop inputs KA	Flip-flop inputs JB	Flip-flop inputs KB
0	0	0	0	0	0	X	0	X
0	0	1	0	1	0	X	1	X
0	1	0	1	0	1	X	X	1
0	1	1	0	1	0	X	X	0
1	0	0	1	0	X	0	0	X
1	0	1	1	1	X	0	1	X
1	1	0	1	1	X	0	X	0
1	1	1	0	0	X	1	X	1







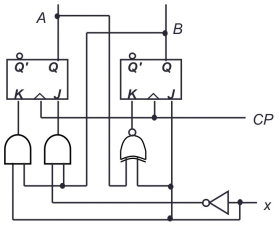


Flip-flop input functions:

$JA = B \cdot x'$ $\quad JB = x$

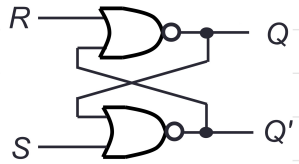
$KA = B \cdot x$ $\quad KB = (A \oplus x)'$

Logic diagram:



S-R

ungated SR latch

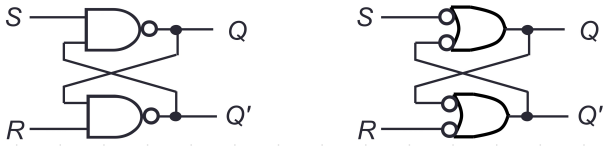


S	R	Q(t+1)	Comments
0	0	Q(t)	No change
0	1	0	Reset
1	0	1	Set
1	1	?	Unpredictable

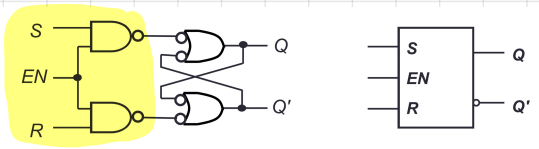
Q	Q^+	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

$Q^+ = ?$

ungated active-low SR latch (inputs are flipped)



gated SR latch / flip flop



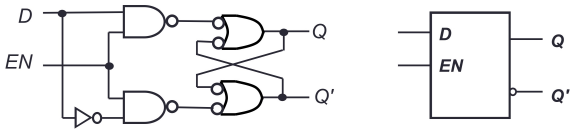
D Direct

gated D latch / flip flop

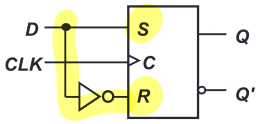
$Q^+ = D$

Q	Q^+	D
0	0	0
0	1	1
1	0	0
1	1	1

D	Q(t+1)	
0	0	Reset
1	1	Set



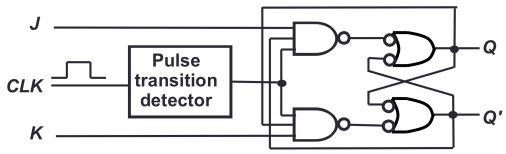
Convert *S-R* flip-flop into a *D* flip-flop: add an inverter.



↳ gets rid of undesirable invalid SR state, but loses ability to 'no change'

JK flip flop

valid version of SR



J	K	Q(t+1)	Comments
0	0	Q(t)	No change
0	1	0	Reset
1	0	1	Set
1	1	Q(t)'	Toggle

Q	Q^+	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

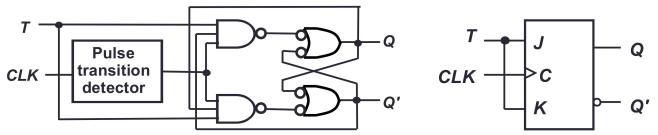
$Q^+ = J \cdot Q' + K' \cdot Q$

↳ has no invalid output. Same as SR, but invalid replaced with toggle.

↳ Q and Q^+ are fed back to pulse-steered NAND gates

T flip flop

Toggle



Q	Q^+	T
0	0	0
0	1	1
1	0	1
1	1	0

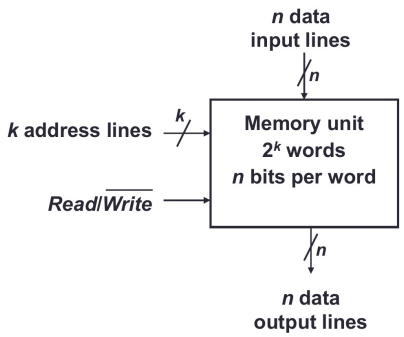
T	Q(t+1)	
0	Q(t)	No change
1	Q(t)'	Toggle

↳ toggles. Formed from JK by tying inputs together.

6 memory

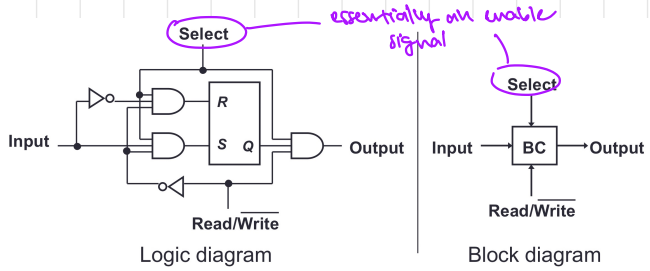
1) the memory unit

Memory Enable	*Read/Write*	Memory Operation
0	X	None
1	0	Write to selected word
1	1	Read from selected word



built from an array of memory cells, with a decoder to select identified memory cells

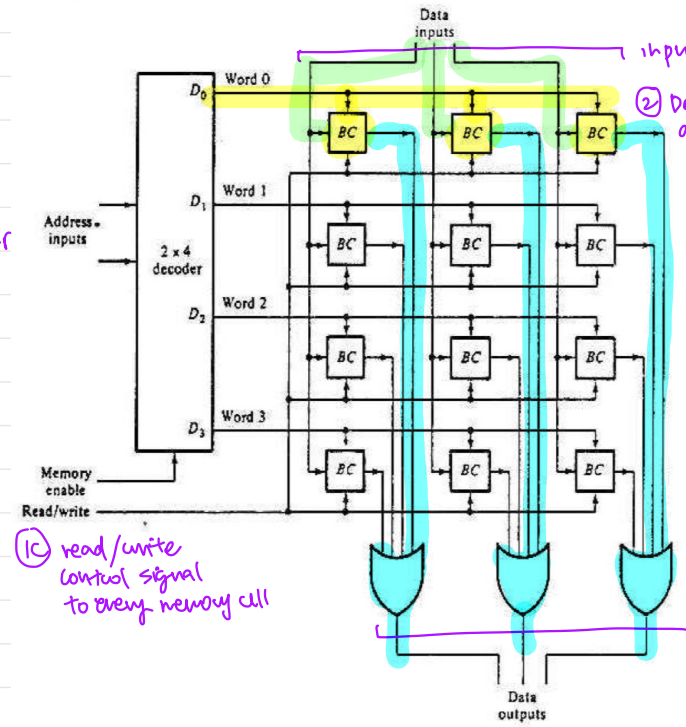
2) memory cells of static RAM : flip flops are used



added wiring around SR flipflop is so that behaviour follows memory control signals

- read = 1 → uncharged
- write = 0 → Q = input
- select → enable

3) arrays of memory cells to build memory



(1A) address into decoder

(1B) memory enable to decoder

(1C) read/write control signal to every memory cell

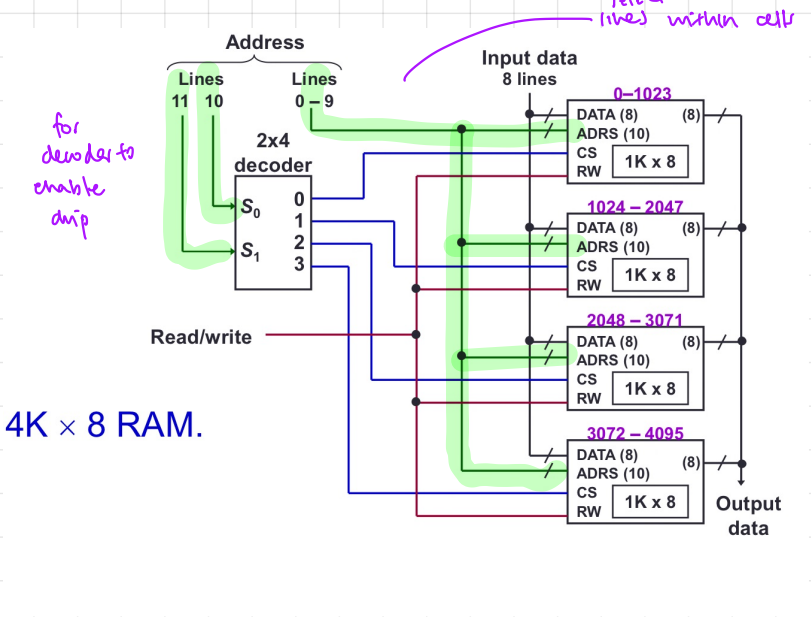
input bits

(2) Decoder enables word determined by address by providing select signal

(3) based on Read/write, state changes / uncharged & output to OR gate

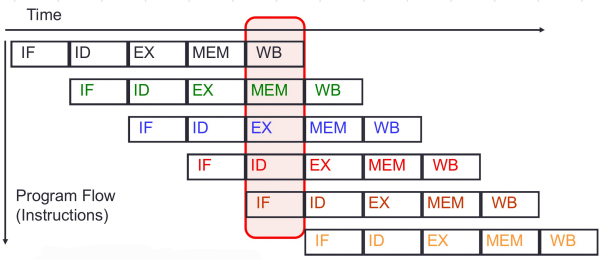
output bits

4) arrays of memory chips to build larger memory



4K × 8 RAM.

13 - Pipelining



intuition: by stacking processes together, we use the hardware more efficiently than we would if we did it sequentially. This improves the throughput of the system. Pipeline rate is slowed by slowest stage due to bottleneck.

IF → ID → EX → MEM → WB

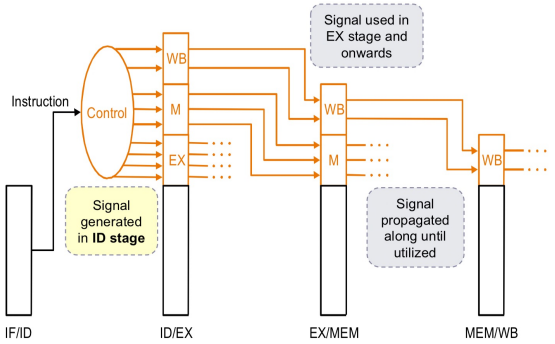
Instruction mem | Reg. file | ALU | memory | Reg. file

① stages and datapath (naive implementation)

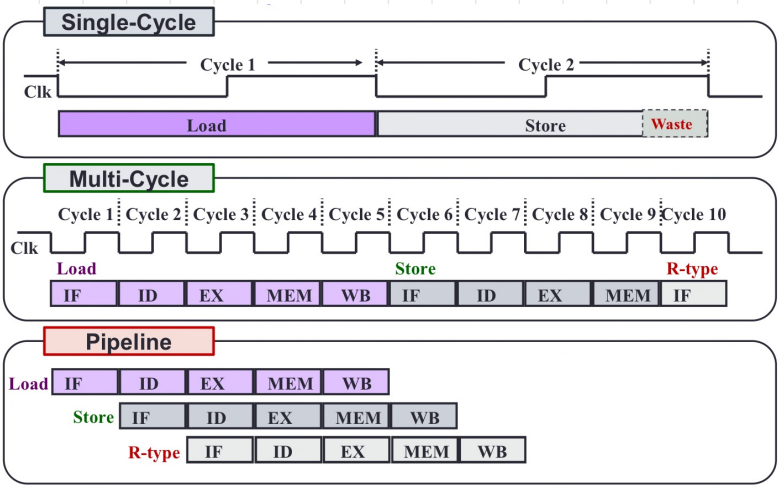
1) **intuition**: to keep instructions separate, we use special 'pipeline registers' to store and pass forward relevant information to the next execution stage

2) **control**: pass on relevant signals through pipeline registers

	EX Stage				MEM Stage			WB Stage	
	RegDst	ALUSrc	ALUop		Mem Read	Mem Write	Branch	MemTo Reg	Reg Write
			op1	op0					
R-type	1	0	1	0	0	0	0	0	1
lw	0	1	0	0	1	0	0	1	1
sw	X	1	0	0	0	1	0	X	0
beq	X	0	0	1	0	0	1	X	0



③ Performance of processors



Single cycle

$CT = \sum T_i$ $\quad T = n_{instructions} \cdot CT$

$cycles = n_{instructions}$

multicycle

$CT = \max(T_i)$ $\quad T = n_{instructions} \cdot average\ CPI \cdot CT$

cycles = variable ⇒ average

pipelined

$CT = \max(T_k) + T_{overhead}$ $\quad T = cycles \cdot CT$

$cycles = n_{instruction} + \underbrace{N-1}_{\text{stages wasted to fill up pipeline}}$

$$Speedup_{pipeline} = \frac{Time_{seq}}{Time_{pipeline}}$$

$$= \frac{I \times \sum_{k=1}^{N} T_k}{(I+N-1) \times (\max(T_k) + T_d)}$$

$$= \frac{I \times N \times T_k}{(I+N-1) \times T_k}$$

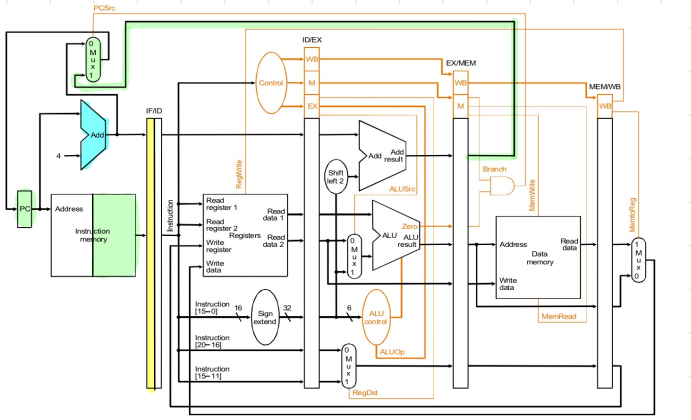
$$\approx \frac{I \times N \times T_k}{I \times T_k}$$

$$\approx N$$

Conclusion:
Pipeline processor can gain **N** times speedup, where **N** is the number of pipeline stages

Naive implementation

1. Instruction fetch



First half

MUX chooses next instruction, passes to IM.

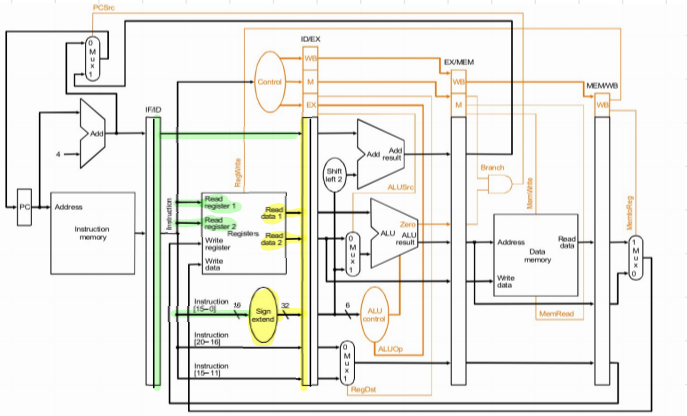
Adder calculates PC+4

Second half

IF/ID register recieves : instruction read

PC+4

2. Instruction decode



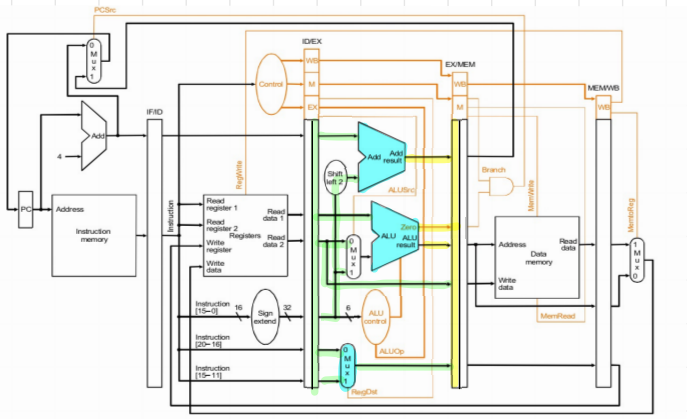
First half

IF/ID supplies register numbers / 16 bit immediate

Second half

Data values read from register file
sign extended immediate
PC+4 pass forward
WR pass forward
} ID/EX recieves

3. Execution



First half

ID/EX supplies : Data values read from register file → ALU

sign extended immediate
PC+4 pass forward
WR pass forward
} PC+4+imm computed

WR or Rt (for SW) chosen by MUX

Second half

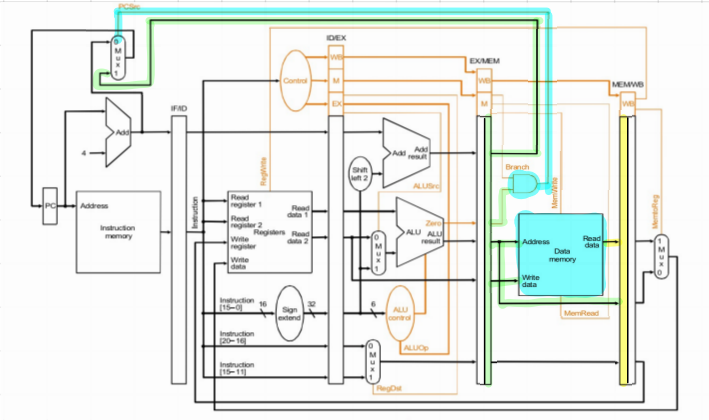
EX/MEM recieves : PC+4 + 4 x imm

ALU result

is Zero? signal

Data read 2 from RF

4. mem stage



First half

EX/MEM supplies : PC + 4 + 4 x imm for next inst.

ALU result for addr.

RD2 writing into mem

ALU result is zero?

memory reads or writes data — rather than in stage 3, to reduce instr. cycle time, since constrain bottleneck of one stage's does too much

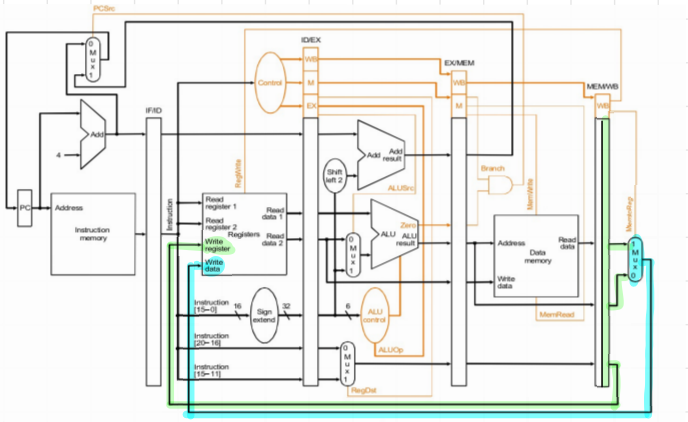
is zero signal used by AND to decide branch

Second half

MEM/WB receives : data from memory

ALU result

5. WB stage



First half

MEM/WB Supplies : ALU result & data from mem

WR register

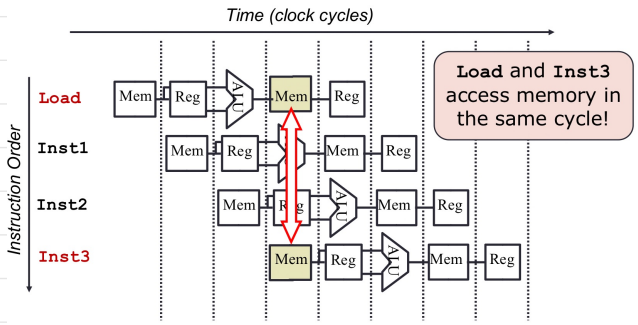
MUX chooses and writes to RF

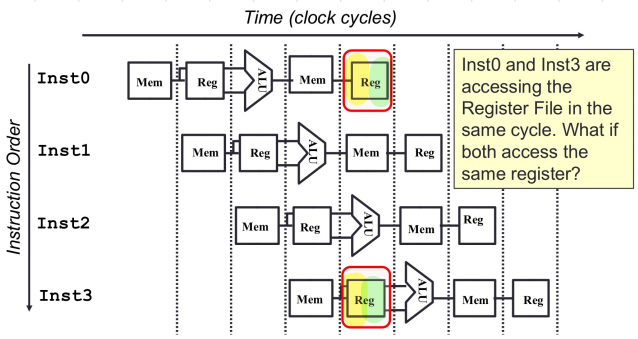
Second half

nothing (RF used by stage 2 to read)

③ structural hazards : conflicts in pipelining that occur when a single piece of hardware requires use by different stages simultaneously

If there is only a **single memory module:**





⇓ split memory into two types : instruction & data

⇓ split cycle into half. first half to write, second half to read

④ Data RAW hazards

RAW affects next 2 instructions

↳ Read After Write (true data dependency): occurs when a later instruction reads from destination register written by an earlier instruction. if read before earlier can write, will erroneously use stale result.

impact: Need to stall

any written ⇒ stage 5
any (read) ⇒ stage 2
} stall 2 (2 3 4 5 ⊗ ⊗ 2 3 ...)
↳ became write in first half of stage 5, read in second half of stage 2

eg.

```
i1: add $1, $2, $3 #writes to $1   R type
i2: sub $4, $1, $5 #reads from $1  R type
```

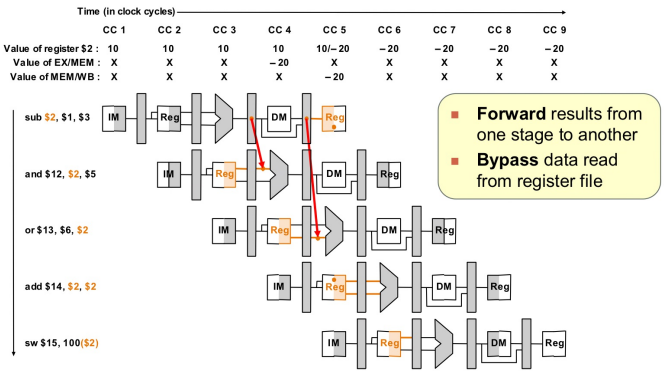
solution : forwarding

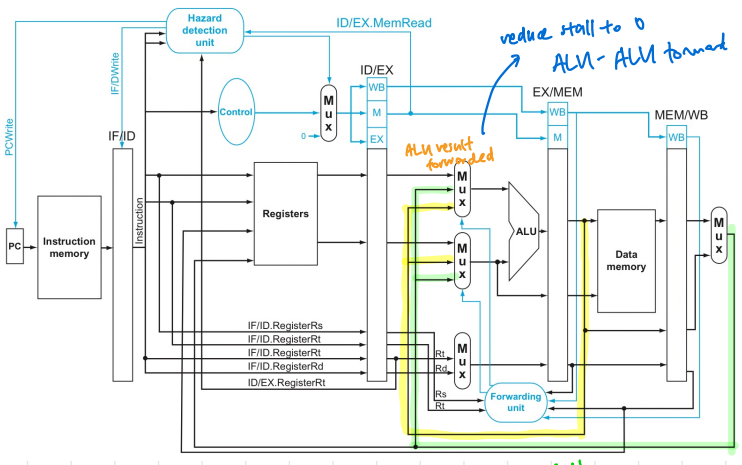
↳ pass the result of computation to the trailing instruction so that it can be used before it is reflected in the register file. This bypasses the data read from the register file.

↳ implementation : connect EX/MEM register output to input of ALU, use hazard detection & forwarding units to provide forwarding control signals. so as MEM reads input from register, ALU (for next instruction) also does at the same time.

↳ if result produced > 1 stage from when it is needed, need to stall

R type, R type ⇒ idea
lw, R type ⇒ stall 1



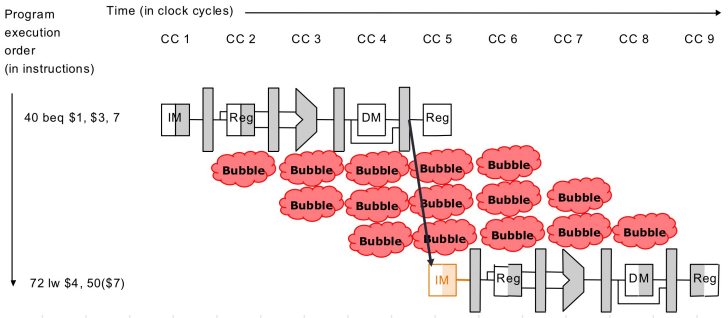


↳ reduce stall to 1
MEM - ALU forward

⑤ Control hazards

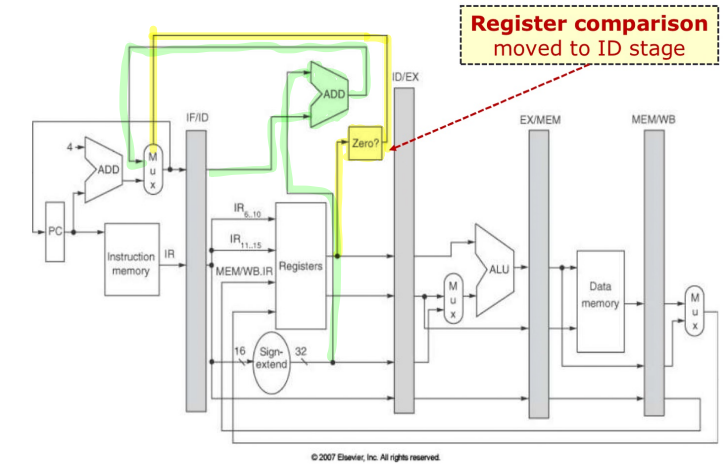
↳ control dependency : j is control dependent on i if i controls whether or not j is executed. Incorrect execution may occur if not managed.

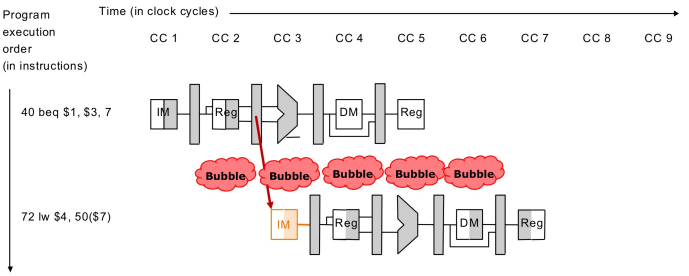
(impact) : branch result produced @ stage 3, but instruction to be fetched/decision only at end stage 4. instruction decode @ stage 1. ⇒ 3 cycle stall



early branch

↳ move comparator from ALU to ID stage and computation of PC+4+4×imm, since ID stage is the earliest this information can be used. Reduces stall to 1 cycle.





↳ problems : also lengthens stage 2. Not good for pipelining

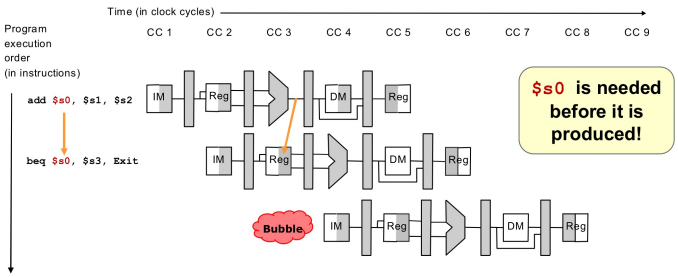
1) RAW in R type

eg. add $s0 $s1, $2 ⇒ $s0 written in stage 5
beq $s0, $3, exit ⇒ branch decision made in stage 2

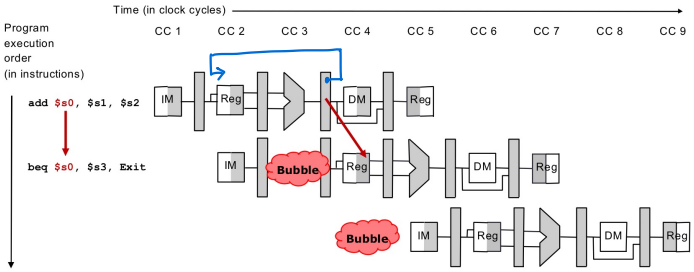
1 cycle stall

⇓

Add forwarding from ALU to ID ⇒ 1 cycle stall



- Add forwarding path from ALU to ID stage
- **One clock cycle delay** is still needed



2) RAW w/ lw

eg.
```
lw $s0, 0($s1)
beq $t0, $s3, exit
```

given early branching implementation, need to implement
MEM → ID forwarding → same no. of stalls as w/o forwarding, but faster

⇒ still 3 stalls required

branch prediction

- Simple prediction:
  - All branches are assumed to be **not taken**
  - → Fetch the successor instruction and start pumping it through the pipeline
- When the actual branch outcome is known:
  - **Not taken**: Guessed correctly → No pipeline stall
  - **Taken**: Guessed wrongly → Wrong instructions in the pipeline → **Flush** successor instruction from the pipeline

easier to implement since PC+4 already computed

if early branching → flush → 1 / 2 RAW w/ R / 3 RAW w/ lw

delayed branch

Observation:
- Branch outcome takes **X** number of cycles to be known
- → **X** cycles stall ⇒ no. of 'slots' to put after branch

Idea:
- Move **non-control dependent instructions** into the X slots following a branch
  - Known as the **branch-delay slot**
- → These instructions are executed **regardless of the branch outcome**

eg.

Non-delayed branch
```
or  $8, $9, $10
add $1, $2, $3
sub $4, $5, $6
beq $1, $4, Exit
xor $10, $1, $11

Exit:
```

Delayed branch
```
add $1, $2, $3
sub $4, $5, $6
beq $1, $4, Exit
or  $8, $9, $10
xor $10, $1, $11

Exit:
```

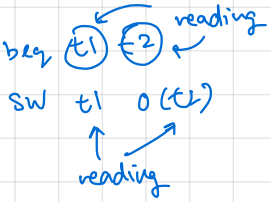
- The "**or**" instruction is moved into the delayed slot:
  - Get executed regardless of the branch outcome
  - → Same behavior as the original code!

↳ either better off or no difference

↳ Not permanent shift of instructions, but just that loop/sequence of execution
eg. first iteration, put initialisation into first beq check

Notes

1. if final instruction has stall → don't include
2. watch out for stalls in loop: RAW cascades to next two, beq last instruction → stall first in next loop
3. jump may be resolved in stage 4 ⇒ like branch. ⇒ 3 cycle stall.
4. when watching out for RAW, be careful w/ instructions. eg. branching



14. caching

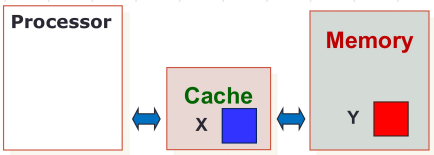
① memory types, locality and caching

i) different types of memory w/ diff. access times, longevity & cost → want high performance at manageable cost → use a mix depending on use case to streamline

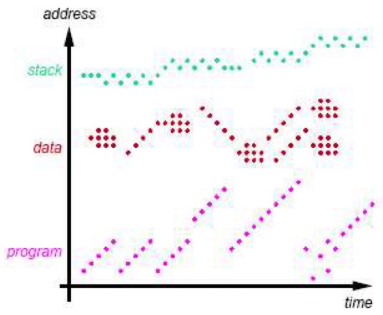
ii) intuition behind caching: to keep frequently / recently used data in smaller and faster memory, only referring to main memory when needed

↓ how to choose what to keep?

principle of locality

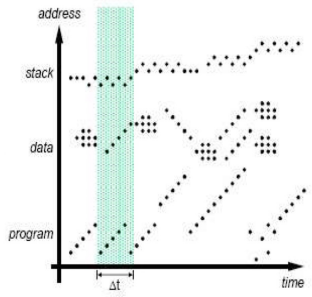


- **Temporal locality**
  - If an item is referenced, it will tend to be referenced again soon
- **Spatial locality**
  - If an item is referenced, nearby items will tend to be referenced soon
- Different locality for
  - Instructions
  - Data



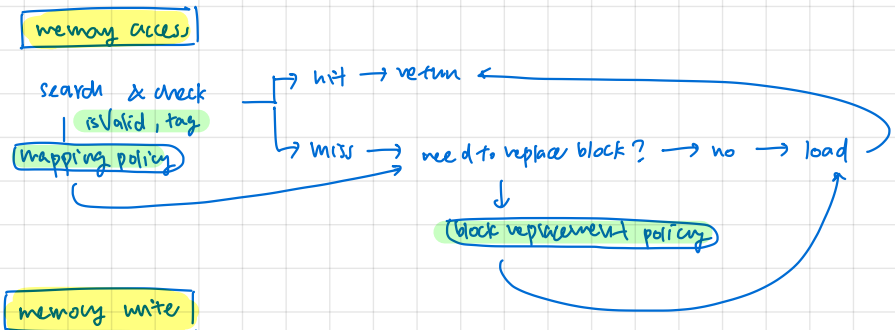
- **Set of locations accessed during Δt**
- Different phases of execution may use different working sets

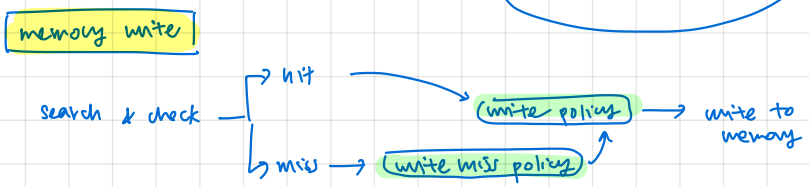
Our aim is to **capture the working set and keep it in the memory closest to CPU**



iii) how caching works: to take advantage of spatial locality, when some data is requested, we also load its nearby data into the cache, as a block.

to take advantage of temporal locality, we determine how previously accessed data is stored or removed through no. of blocks, mapping & replacement policy





4) hits & misses

Hit: Data is in cache (e.g., **X**)
- Hit rate: Fraction of memory accesses that hit
- Hit time: Time to access cache

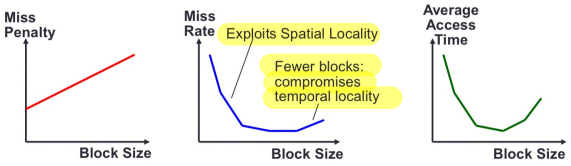
Miss: Data is not in cache (e.g., **Y**)
- Miss rate = 1 – Hit rate
- Miss penalty: Time to replace cache block + hit time

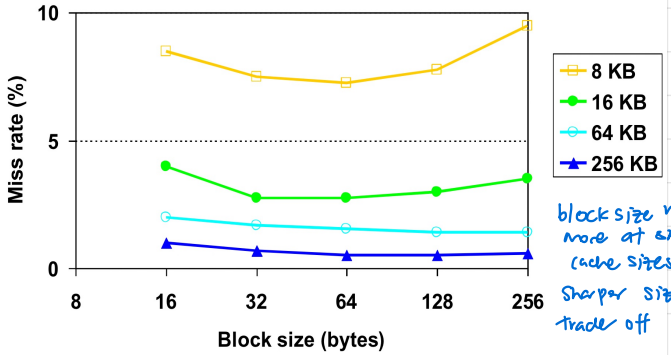
Average Access Time
= Hit rate x Hit Time + (1-Hit rate) x Miss penalty

- **Compulsory misses** (ALL)
  - On the first access to a block; the block must be brought into the cache @ first call
  - Also called cold start misses or first reference misses
- **Conflict misses** (SA, direct)
  - Occur in the case of direct mapped cache or set associative cache, when several blocks are mapped to the same block/set i.e. something air there
  - Also called collision misses or interference misses
- **Capacity misses** (ALL)
  - Occur when blocks are discarded from cache as cache cannot contain all blocks needed
    i.e. was once there, but removed due to other call ⇒ cache size dependent
    size ↑, miss ↓

2) block size - no. trade off

- Larger block size:
  - \+ Takes advantage of spatial locality
  - – Larger miss penalty: Takes longer time to fill up the block
  - – If block size is too big relative to cache size
    - ➔ Too few cache blocks ➔ miss rate will go up





block size matters more at small cache sizes due to sharper size – no. trade off

3) write policy (sw instructions)

write request → miss
↓
hit → write to cache

Cache and main memory are inconsistent
- Modified data only in cache, not in memory!

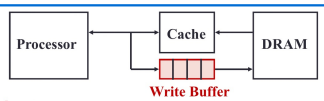
Solution 1: Write-through cache
- Write data both to cache and to main memory

Solution 2: Write-back cache
- Only write to cache
- Write to main memory only when cache block is replaced (evicted)

Write Miss option 1: Write allocate
- Load the complete block into cache
- Change only the required word in cache
- Write to main memory depends on write policy

Write Miss option 2: Write around
- Do not load the block to cache
- Write directly to **main memory only**

write through cache (both)



Problem:
- Write will operate at the speed of main memory!

Solution:
- Put a write buffer between cache and main memory
  - Processor: writes data to cache + write buffer
  - Memory controller: write contents of the buffer to memory (kinda like async)

write back cache (cache, memory @ evictions)

Problem:
- Quite wasteful if we write back every evicted cache blocks

Solution:
- Add an additional bit (**Dirty bit**) to each cache block
- Write operation will change dirty bit to 1
  - Only cache block is updated, no write to memory
- When a cache block is replaced:
  - Only write back to memory if dirty bit is 1 ⇒ bit reset

↳ so writing from cache to memory only when necessary

④ Mapping policy & Cache types

Direct mapped cache

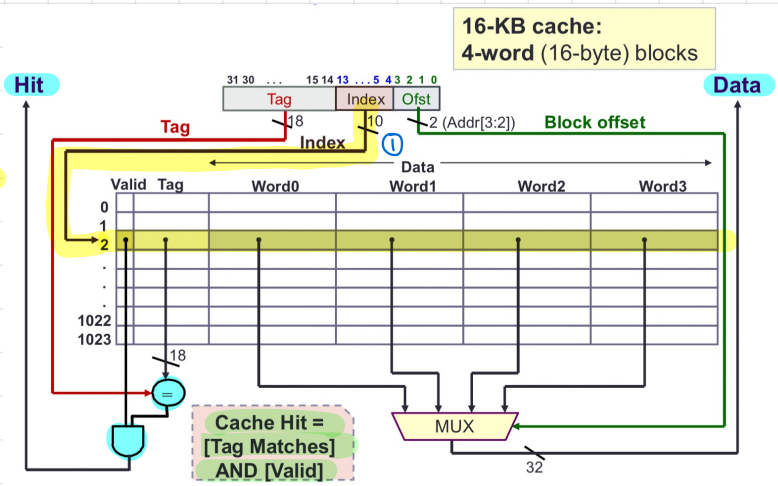
how it works one set, one word.

↳ cache is made up of 2^M distinct memory locations each able to hold one block of 2^N bytes.

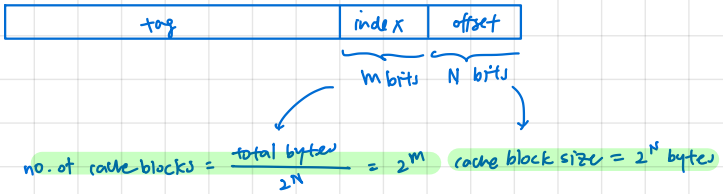
↳ so we map blocks with the same 2^M (index) bits to the same block, and use their last 2^N bits (offset) to differentiate them within the block.

↳ then we use the remaining bits (tag) to check for uniqueness during hits.

implementation



encoding



① index bits as input for MUX to select cache row.

② unique tag of request and in cache compared using comparator. Then AND with is Valid to see if hit → return / replace

③ offset bits as input for MUX to select byte(s)

Set Associative Cache

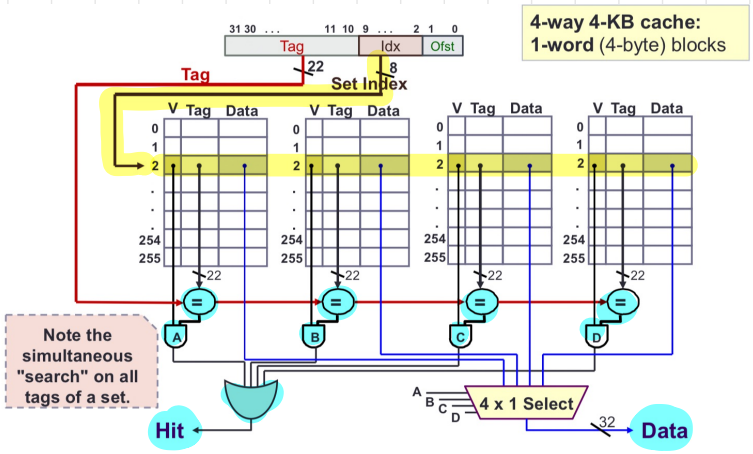
how it works

↳ n-way: cache made up of distinct rows of sets, each set able to hold n blocks
⇒ reduce conflict misses

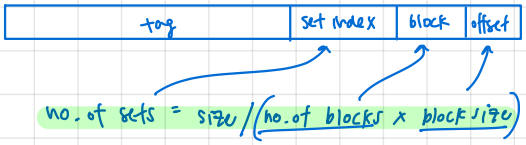
↳ map data to sets and then blocks using index and 'two-part' offset

↳ check requires check on all memory cells in set simultaneously

implementation



encoding



① index bits choose set, simultaneous is Valid / tag comparison for each block, then OR, to see if hit

② offset no. used by MUX to select data within blocks

③ block no. used by MUX to select pre-selected data as output

---

Fully associative cache

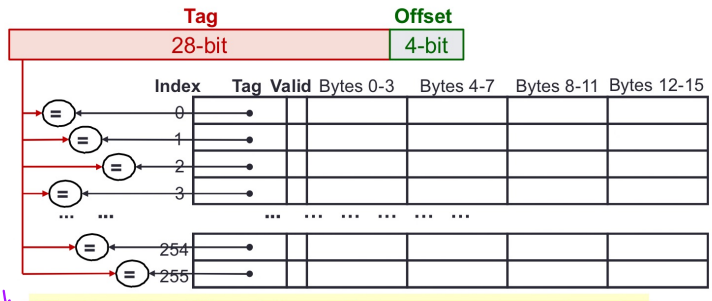
how it works

↳ no mapping policy. Just added.

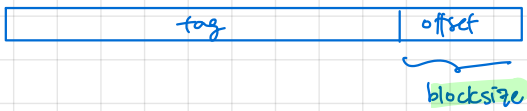
↳ so implementation requires simultaneous check on ALL memory cells

implementation

- Example:
  - 4KB cache size and 16-Byte block size
  - Compare tags and valid bit in parallel



encoding



★ **No Conflict Miss (since data can go anywhere)**

⑤ Comparing cache types (balancing complexity & hit rate)

Direct

pros: simple to implement, fast

cons: ↳ heavy block size - number trade off
↳ a lot of conflict misses
↳ cannot handle alternating calls

Fully associative cache

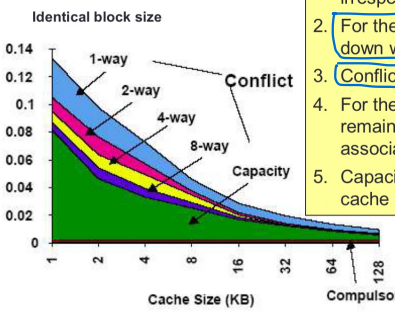
pros: eliminates conflict misses

cons: ↳ capacity misses occur later on
↳ implementation is very complex, greater overhead to search w/o indexing + larger tags to compare

Set associative

↳ balance of both direct and fully associative - less conflict misses, slightly more complex implementation

↳ Direct cache of size N has same miss rate as 2-way SA of size N/2



Observations:
1. Cold/compulsory miss remains the same irrespective of cache size/associativity.
2. For the same cache size, conflict miss goes down with increasing associativity.
3. Conflict miss is 0 for FA caches.
4. For the same cache size, capacity miss remains the same irrespective of associativity.
5. Capacity miss decreases with increasing cache size .

Total Miss = Cold miss + Conflict miss + Capacity miss

Capacity miss (FA) = Total miss (FA) – Cold miss (FA), when Conflict Miss → 0

⑤ Block replacement policy (SA or FA caches)

↳ SA and FA caches can 'choose' where to place new block, potentially replacing another block if full

↳ how to choose?

Least recently used (LRU)

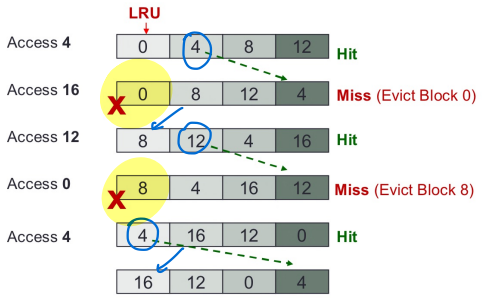
- capitalises on temporal locality
- for every cache access, record which block in a queue

} difficult to implement if many subblocks & ways

others

Other replacement policies:
- First in first out (FIFO)
- Random replacement (RR)
- Least frequently used (LFU)

Least Recently Used policy in action:
- 4-way SA cache
- Memory accesses: **0 4 8 12 4 16 12 0 4**



Notes

Block Placement: Where can a block be placed in cache?

Direct Mapped:	N-way Set-Associative:	Fully Associative:
• Only one block defined by index	• Any one of the **N** blocks within the set defined by index	• Any cache block

Block Replacement: Which block should be replaced on a cache miss?

Direct Mapped:	N-way Set-Associative:	Fully Associative:
• No Choice	• Based on replacement policy	• Based on replacement policy

Block Identification: How is a block found if it is in the cache?

Direct Mapped:	N-way Set Associative:	Fully Associative:
• Tag match with only one block	• Tag match for all the blocks within the set	• Tag match for all the blocks within the cache

Write Strategy: What happens on a write?

Write Policy: Write-through vs write-back

Write Miss Policy: Write allocate vs write no allocate

1. Take note of block replacement. If instructions not actually called (eg. branch) then even in loop, may not overwrite. ⇒ first loop cold miss, after all hit

2. Take note of order of access in seeing replacement ⇒ decide hits (see T11 Q3 G)
   ↳ if clash, but excess from middle onwards ⇒ the 'non middle' will prevail.

eg. A(0) _ C(0) _ ⇒ A(0) A(1) A(2) A(3)
C(1) ← C(2) doesn't clash

3. calculating array address. A[2] = &A[0] + 4×2

4. associative cache: just put. No block index.

5. counting hit rate: watch out for lw & sw. if load store → 2 accesses.
   ↑ ↑
   1 access each.

8. 64 words 1 block 4 words

index → 4 bits
offset → 4 bits
4 bits → 2 bit word, 2 bit byte

9. 1 set = group of blocks ⇒ row in FA cache. But each block only 1 tag series of bits!
   NOT each word 1 tag!