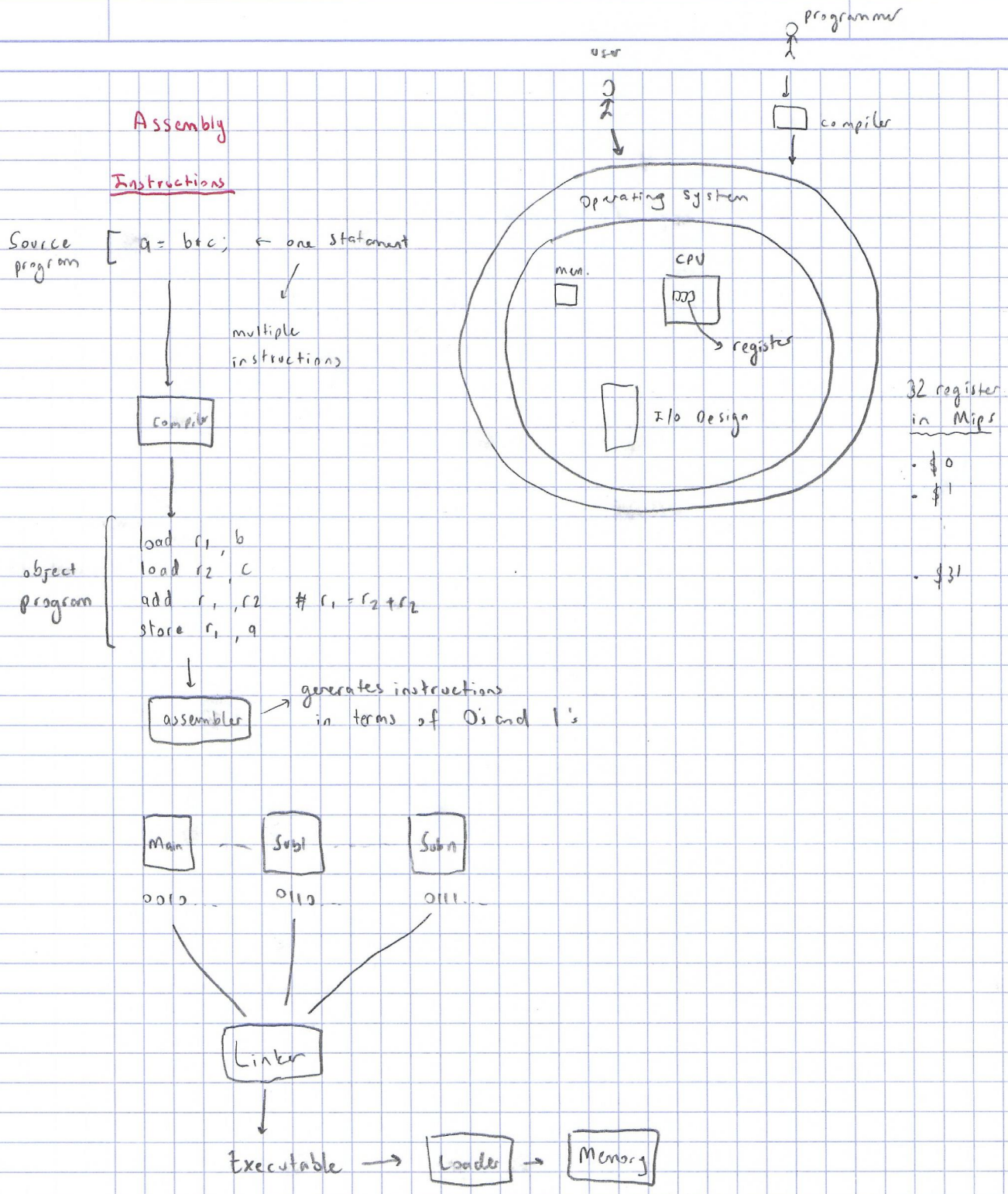
Assembly

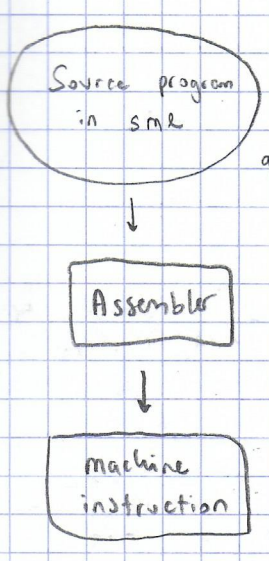
Instructions



32 register in Mips

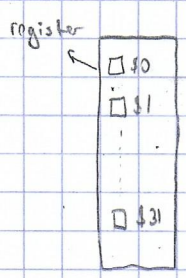
- $0
- $1

- $31



sml : sembolic machine language

add r_1, r_2

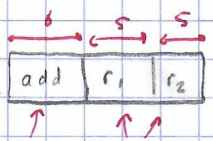


$0 → first register = 00000

$31 → last register = 11111

- add r_1, r_2 → register

Assume that result stored at r_2 ($r_1 = r_1 + r_2$)



→ 16 bits = 2 bytes

operator — operands

- In mips all instructions have same bits. However, for ex in intel all instructions have different bits.

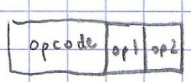
Machine Instructions

MIPS : all instructions have the same size. = 32 bits = 4 bytes

Types:

- R : all operands are registers
- I : register and the memory operand
- Jump Instructions : Memory adder that we want to jump.

Hypothetical two adress machine.



no of bits	4	2	2
	0000	00	00
	:	01	01
	:	10	10
	1111	11	11

↓

at most 16 operations

- $r_1 = r_2 + r_3$

```
add r2 , r3  # r2 = r2 + r3   } we lost r2 !
move r1 , r2 # r1 ← r2        }
```

```
move r1 , r2 ──→ [assembler] ──→ opcode  op1  op2
add  r1 , r3                     1101    01   10
                                 0001    01   11
```

Hexa
D6
17

machine operation code

mnemonic	opcode
add	0001
move	1101

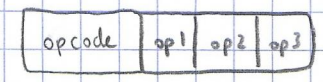
↓

[Disassembler]

↓

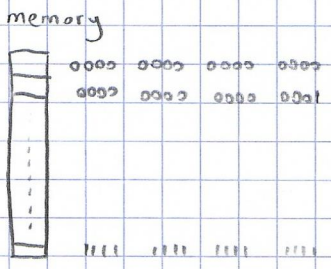
```
move r1 , r2
add  r1 , r3
```

Consider a hypothetical 3-adress machine



8 bits | 16 | 16 | 16 ⇒ 7 bytes

they refer to memory locations



Memory Size
16 bits
↓
$2^6 \times 2^{10}$
↓ ↳ 1024 → 1 Kilo
64
→ 64 KB

add a , b , c

↳ Don't Forget, it's in hexadecimal

→ Assembler Passes

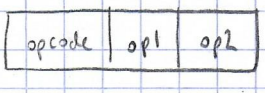
- 1st pass: Generates the symbol table.

symbol	Mem location
a	0020
b	
c	

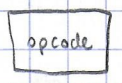
- 2nd pass : Generates the object program

Complete assembly process

- 0 adress machine ← we have two types of instructions
- 2 adress machine

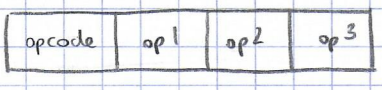


0 - adress

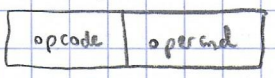


8 bits

- 3 adress machine



1- adress



8 bits 16 bits

Machine operation Table (MOT)

(Hexa) opcode	Mnemonic	operand	explanation
01	load	A	push A
02	store	A	pop → top goes to A
03	add	-	SL ← SL + TL , pop
04	sub	-	SL ← SL - TL , pop
05	mult	-	SL ← SL * TL , pop
06	div	-	SL ← SL / TL , pop
07	stop	-	

SL → second Level
TL → Top level

It's like a Stack.

a = b + c * d + e * f

(Hexa) location	machine inst	Symbolic machine instruction	
0 (0)	01 00 17	load b	| b | - - -
3 (3)	01 00 1B	load c	| b | c | - - -
6 (6)	01 00 1F	load d	| b | c | d | - - -
9 (9)	05	multiply	| b | c * d | - - -
10 (A)	01 00 23	load e	| b | c * d | e | - - -
13 (D)	01 00 27	load f	| b | c * d | e | f | - - -
16 (10)	05	multiply	| b | c * d | e * f | - - -
17 (11)	03	add	| b | c * d + e * f | - - -
18 (12)	03	add	| b + c * d + e * f | - - -
19 (13)	02 00 2B	store a	
22 (16)	00	stop	

NOTE: block word → blkw → int a; ← without initializing
(for next page) word 5 → int b = 5; ← with initializing

M/c : machine

location	machine inst	Symbolic m/c instructions
00 00 00 00 ← 23 (17)	00 00 00 05	b: .word 5
00 00 00 0A ← 27 (1B)	00 00 00 0A	c: .word 10
00 00 00 0F ← 31 (1F)	00 00 00 0F	d: .word 15
00 00 00 14 ← 35 (23)	00 00 00 14	e: .word 20
00 00 00 19 ← 39 (27)	00 00 00 19	f: .word 25
00 00 00 00 ← 43 (2B)	00 00 00 00	a: .blkw 1
47		end

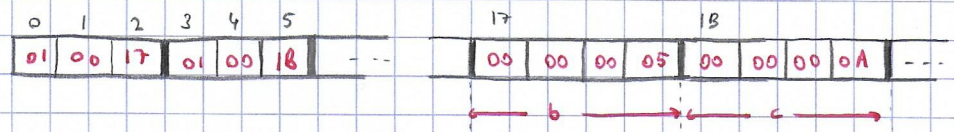
for an array of 10 elements
a: .blkw 10

physical end for the program

Symbol Table

Symbol / Variables	Value / Address (Hex)
b	17
c	1B
d	1F
e	23
f	27
a	2B

Load process into main memory



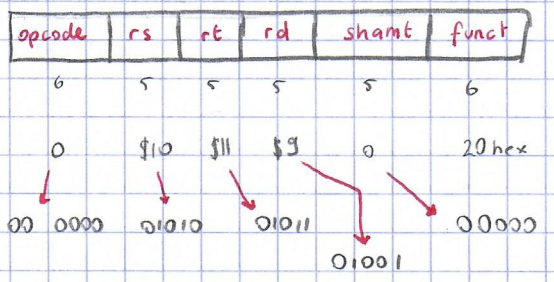
- Absolute Program: Must be loaded into the same location.
- Relocatable Program: Can be loaded anywhere

Example: add $t1, $t2, $t3

add $t9, $t10, $t11

rd — destination; rs — second; rt — third

R-type →



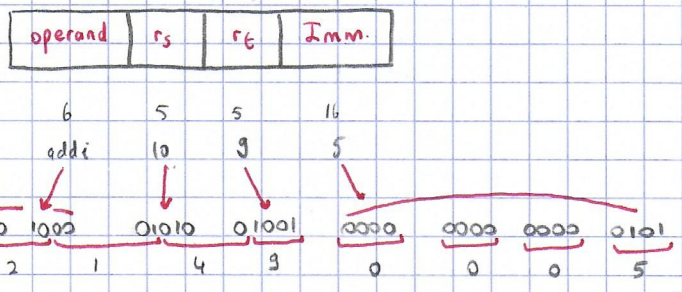
Example: addi $t1, $t2, 5 # $t1 = $t2 + 5

addi $9, $10, 5

rt rs

R[rt] = R[rs] + SignExtendedImmediate Data

I



=> 21 49 00 05 16

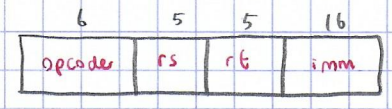
Example: ori $t1, $t2, -8

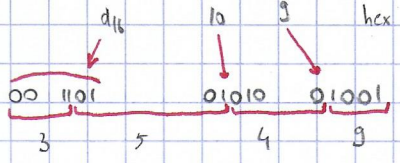
$t1 = $t2 | -8

-8 => 01000 → Reverse and add 1 → 10111 + 1 => 11000

8 — for being safe

11000 — -8 in two's complement





hex → FF F8

=> 35 49 FF F8 (?)

Example: Write a program which will do q = b + c;

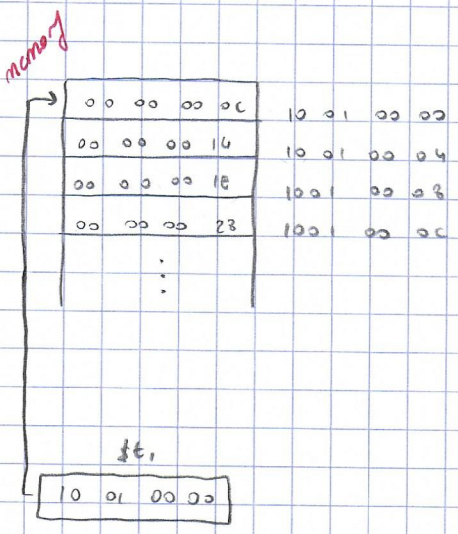
```
        .text
        .globl  -- start

-- start :

        lw   $t1 , b
        lw   $t2 , a
        add  $t3 , $t1 , $t2
        sw   $t3 , c
        li   $v0 , 0
        syscall


        .data
a:  .word 10
b:  .word 20
c:  .word 0
        .end
```

Example:

```
        .data
array : .word   12, 20, 30, 40
```

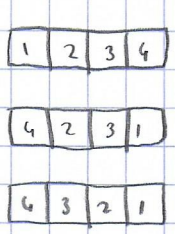
This part needs to be at last

```
        .text
        .globl  -- start

-- start :

        li   $t2 , 0    #  $t2 = sum;
        la   $t1 , array
        lw   $t3 , 0 ($t1)
        add  $t2 , $t2 , $t3   # $t2 = $t2 + $t3

        lw   $t3 , 4 ($t1)
        add  $t2 , $t2 , $t3

        lw   $t3 , 8 ($t1)
        add  $t2 , $t2 , $t3

        lw   $t3 , 12 ($t1)
        add  $t2 , $t2 , $t3

# stop

        li $v0, 0   } This 2 lines stops the program.
        syscall
```




```
        .data
        .
        .
        .end
```

Exchange Array Elements

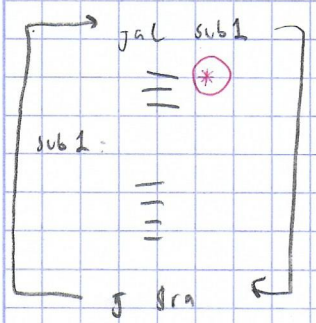



```
        la   $t1, array
        lw   $t2, arraysize
        subi $t2, $t2, 1
        mul  $t2, $t2, 4
        add  $t2, $t2, 1
next:
        bge  $t1 $t2 done

        lw   $t3, 0, ($t1)
        lw   $t4 , 0 ($t2)

        sw   $t4 , 0 ($t1)
        sw   $t3 , 0 ($t2)

        addi $t1 , $t1, 4
        subi $t2 , $t2, 4

        b next
```

Functions:

```
        li   $s0, 5
        li   $t1 , 5
```

(*) $s0 is guaranteed to be equal to 5
$t0 can have a different value.



E.x: re max(x, y) + max(v, w)

```
        .data

x:      .word 10
y:      .word 20
v:      word 30
w:      .word 40
result: .word 0
```

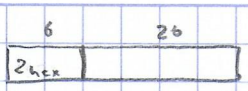
```
main:

        lw $a0, x
        lw $a1, y
        lw $a2, v
        lw $a3, w

        jal sum   # Sum returns results in $v0

        sw $v0, result

        # Print sum
        # Stop
```

```
sum:

        addi $sp, $sp, -16
        sw   $ra, 12($sp)
        sw   $s0, 8($sp)
        sw   $s1, 4($sp)
        sw   $s2, 0($sp)

        jal  max

        move $s0, $v0  # $s0 = max(x,y)
        move $a0, $a2
        move $a1, $a3

        jal  max

        move $s1, $v0
        add  $s2, $s0, $s1
        move $v0, $s2  # $v0 holds the result

# Restore initial values of $ra, $s0, $s1, $s2

        lw   $s2, 0($sp)
        lw   $s1, 4($sp)
        lw   $s0, 8($sp)
        lw   $ra, 12($sp)
        addi $sp, $sp, 16
        jr   $ra


max:
        move $t0, $a0
        move $t1, $a1

        bgt  $t0, $t1, next

        move $v0, $t1
        b    done

next:
        move $v0, $t0

done:
        jr   $ra
```

Here is save ra, s0, s1 & s2

We cannot take the last 2 bits too

L0: J L3 0040 0000
add ... 0040 0004
add ... 0040 0008



=> 00 0010 | 0000 0100 0000 11
0 4 0 C

L1: J L0 0040 000C
add 0040 0010

The location we want to Jump → 0040 000C

0040 000C

Jump does not store the first 4 bits of PC (Program Counter)

NOTE: Instructions are always word boundry In other words, their address is divisible by 4 → Last two bits are 00

00 0010 0000 0100 0000 0000 0000 0000

→ Machine Code: 08 10 00 00 03

Example: Write a sub program to count no. of occurances of an ASCII characters.

Stored in $a1

$a0 return result $v0

How to set characters from the Array

lbu $t0, 0($a0)

Example: (Slide 106) Factorial: $n! = n * (n-1)!$

$(n-1) * (n-2)! \rightarrow \dots \rightarrow 2 * (1!)$ (1! = 1)

factorial:

```
addi $sp , $sp , -8
sw   $a0 , 4 ($sp)
sw   $ra , 0 ($sp)
```

base case: (next page)

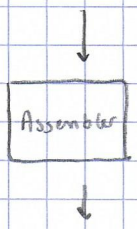
set less than # check if it is in the base case

```
add  $t0 , $0 , 2
slt  $t0 , $a0 , $t0
beq  $t0 , $0 , else
```

```
    addi   $v0, $0, 1
    addi   $sp, $sp, 8
    jr     $ra

else:
    addi   $a0, $a0, -1   # n = n-1
    jal    factorial      # recursive call

    lw   $a0, 4($sp)
    mul  $v0, $a0, $v0
    lw   $ra, 0($sp)
    addi $sp, $sp, 8
    jr   $ra
```

Ex:

- nop → sll $0, $0, $0
- move $t1, $t2 → add $t1, $0, $t2
- blt $t1, $t2, L1




```
slt  $at, $t1, $t2
bne  $at, $0, L1
```

- bgt $t1, $t2, L2


```
slt  $at, $t2, $t1
bne  $at, $0, L2
```

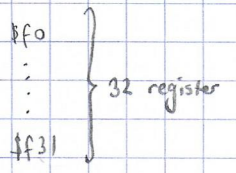
Floating Point Numbers

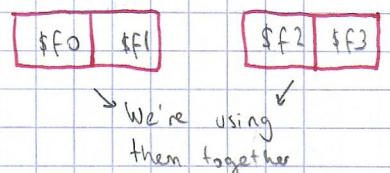
Representation → single precision 4 bytes
→ double precision 8 bytes

Accuracy problems

Floating Point Instructions → have a fp processor.

They have their own register:



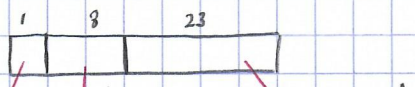


Representation of FP Numbers:

Single Precision: First represent the number in binary.

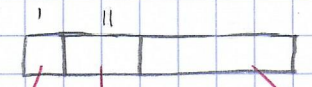
→ We will use IEEE 754 standard for FP representation

Single Pre:



- sign: 0: + ; 1: -
- Exponent: Excess 127; 127 = $7F_{16}$
- significand

Double Precision:



- sign
- exponent: excess 1023; 1023 = $3FF_{16}$
- fraction

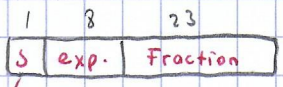
Examples: $9.0_{10} \Rightarrow 1001.0_2 \Rightarrow$ write in scientific notation

$$1.001_2 \times 2^3$$

Exponent = 127 + 3 = 130 ∧

$$7F_{16} + 3_{16} = 82_{16}$$

Example:



0 | $1000\ 0010_2 \rightarrow 82_{16}$ | 001 000... (we add zero's to the end.) $\Rightarrow 41\ 10\ 00_{16}$

NOTE: $80.5_{10} \Rightarrow X_{16} = 50.8_{16}$ (convert directly)

0.5 × 16 = 8.0 → when we see 0, we need to stop

NOTE: $80.55_{10} \Rightarrow X_{16} = 50.8CCCC\ldots$ → 50.8_{16}

.55 × 16 = 8.8

.8 × 16 = 12.8

.8 × 16 = 12.8

:

13/02/2017, Monday

Ex: $178 \Rightarrow X_{16}$

$\frac{178}{16} = 11$ Remainder 2

$\frac{11}{16} = 0$ Remainder 11

$178 = B2_{16}$

$11 \times 16^1 + 2 \times 16^0 = 178$

Fractional Part: $0.140625 \Rightarrow X_{16}$

$0.140625 \times 16 = 2.25$

$0.25 \times 16 = 4.0$

So, $0.140625 = 0.24_{16}$

$2 \times 16^{-1} + 4 \times 16^{-2}$

$178.140625 = B2.24_{16} \Rightarrow 1011\ 0010\ .\ 0010\ 0100$

$1011\ 0010\ 0010\ 01 \times 2^7$

Normalized Form.

NOTE: Floating point no. representation is IEEE 754 Standart

SP → Single Precision

S	Exp	Fraction

31 30 23 22 0

1 → 8 → 23 →

excess $127 = 7F_{16}$

$7F + 7 = 86_{16}$

S = 0 ⇒ +

S = 1 ⇒ −

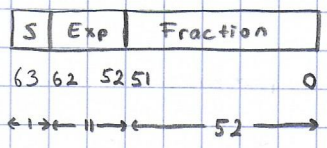
S | exp | Fraction

0 | 1000 0110 | 01100100010010000000

4 3 3 2 2 4

$\Rightarrow 43224_{16}$

Double Precision



$$3FF + 7 = 406_{16}$$

Bias = 1023 = $3FF_{16}$

S — Exp

0 100 0000 0110 0111 0010 0010 1 0

4 0 6 6 4 5

$$40\ 66\ 45\ 00\ 00\ 00\ 00\ 00_{16}$$

Accuracy Problem in Floating Numbers:

Add 9.999×10^{1} 1.610×10^{-1}

1- Align the decimal point of numbers with the larger exponent.

9.999×10^{1} 1.610×10^{-1} $\Rightarrow$ $0.1610 \times 10^{0} = 0.0161 \times 10^{1}$

2- Add the new form of the numbers

$$\begin{array}{r} 9.999 \times 10^{1} \\ +\ 0.016 \\ \hline 10.015 \times 10^{1} \end{array}$$

3- Normalize the result

$10.015 \times 10^{1} = 1.0015 \times 10^{2}$

4- Round result if required

1.002×10^{2}

Binary Addition Using scientific notation.

$(1.000_2 \times 2^{-1})$ $(-1.110_2 \times 2^{-2})$

1- Allign the binary point of the numbers with the larger exponent

$1.110 \times 2^{-2} \Rightarrow 0.111 \times 2^{-1}$

2- Add the numbers

$$\begin{array}{rl} 1.000 & \times 2^{-1} \\ -0.111 & \times 2^{-1} \\ \hline 0.001 & \times 2^{-1} \end{array}$$

16/03/2017 Persembe

3- Normalize the result

$0.001 \times 2^{-1} \Rightarrow 1.000 \times 2^{-4}$

Binary Floating Point Multiplication.

1.000×2^{-1} * -1.110×2^{-2}

1- Add Exponents

$(-1) + (-2) = (-3)$

2- Multiply Numbers

$$\begin{array}{r} 1.000 \\ \times\ 1.110 \\ \hline 0000 \\ 1000 \\ 1000 \\ 1000 \\ \hline 1.11000 \end{array}$$

3- Normalize → 1.110

4- No need to change exponent

1.110×2^{-3}

5- Sign bits are different we have - result

-1.110×2^{-3}

CHAPTER 7

- Micro architecture

↑

parts & how they are connected.

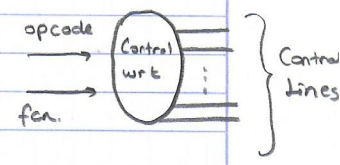
Data Path Components

- Instruction Memory
- Register file: contains 32 register $0 - $31
- Data memory

↑

State elements

- Control lines
  - Reg. Destination
  - Mem Write
  - Reg Write



add rd, rs, rt
lw rt, imm(rs)

immediate value → 16 bits.
There is another component that extending bits to 32.

- Multiplexers

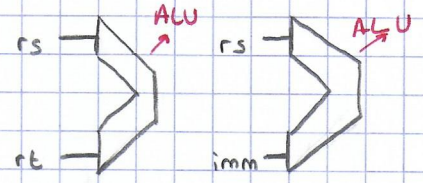
→ We have 4 multiplexers.

i) add rd, rs, rt
lw rt, imm(rs)

RF

↑ which register to write for this, we need multiplexer.

ii)



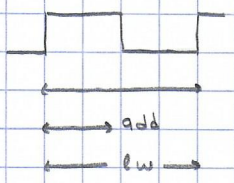
iii) add rd, rs, rt
lw rt, imm(rs)

iv) PC = PC+4

PC ← BTA coming from beq inst.

Data Path Implementation.

1- Single cycle



2- Multicycle : for different instructions we have different number of clock cycles.

3- Pipe lining → stages during inst. execution.

IF = Instruction Fetch
ID = Instruction Decode : get values of registers understand the purpose of instruction.

E.x: Execution.

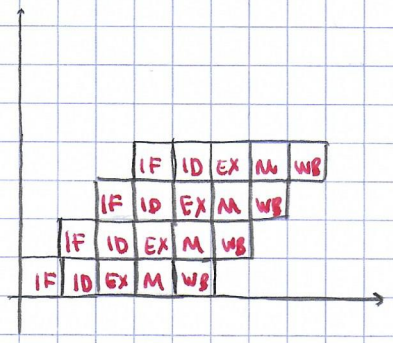
```
add rd, rs, rt
lw  rt, imm(rs)
```

M: Access memory in the lw, sw

WB: Write Back

In Pipelining we have instruction level parallelism.

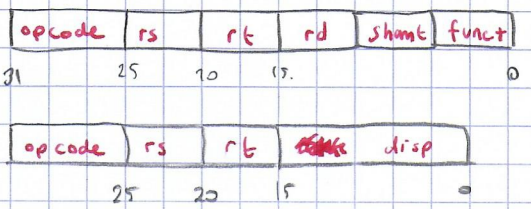
Instruction No

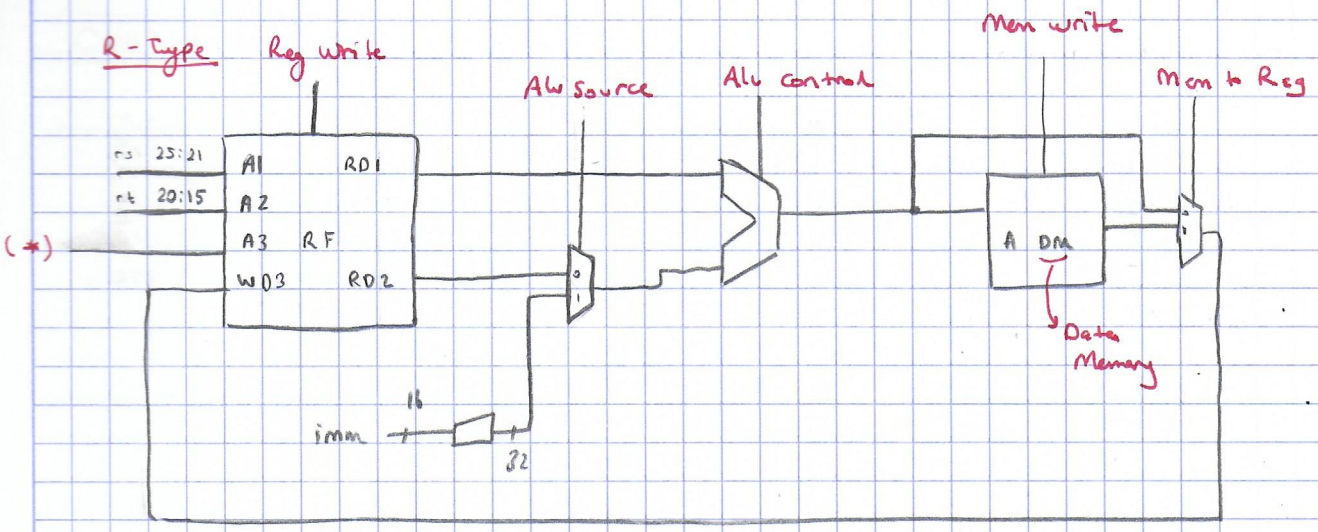



```
add $t1, $t2, $t3
sub $t0, $t1, $t2
```

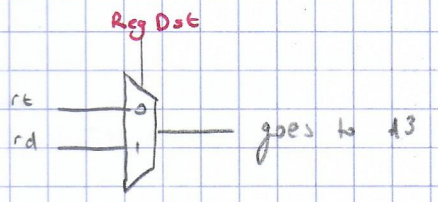
DataPath for single cycle

```
lw  rt, disp(rs)
add rd, rs, rt
```





→ Before (*) is above



20/03/2017, Monday

RTL: Register Transfer Language

DM: Data Memory

IM: Instruction Memory

- add rd, rs, rt
  InstMem[PC] // fetch Instruction
  RF[rd] ← RF[rs] + RF[rt]
  PC ← PC+4

- ori rt, imm
  IM[PC] // fetch Instruction
  RF[rt] ← RF[rs] or zeroExt(imm)
  PC ← PC+4

- lw rt, imm(rs)
  IM[PC]
  RF[rt] ← DM[RF[rs] + signExt(imm)]
  PC ← PC+4

- sw rt, imm(rs)
  IM[PC]
  DM[RF(rs) + SignExt(imm)] ← RF[rt]
  PC ← PC+4

- Assume that we have swap

  swap rs, rt // hypothetical instruction

  RF[rs] ← RF[rt]

  RF[rt] ← RF[rs]

- beq rs, rt, label

  RF = R → in some book we have R to show RF (Register File)

  IM[PC]

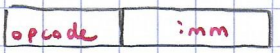
  if (R[rs] - R[rt] eq 0)

  PC ← PC +4 + signExt (imm) * 4

  else

  PC ← PC +4

- j label



  IM [PC]

  PC ← PC[31:28] || imm[26] || 00

E.x: Hypothetical Instruction

sw+ $t1, 0($t2) // sw+ rt, imm(rs)

IM[PC]

DM[RF[rs] + SignExt (imm16)] ← RF[rt]

RF[rs] ← RF[rs] +4

PC ← PC +4

→ We moved the slides

→ After the slides

CPI: Clock cycle / inst. → Single cycle => CPI = 1

for execution time we use this formula

Program execution time = (NO. of instructions) x (clock cycle / inst) x clock cycle time

CPI

- Clock rate 2 GHz = 2×10^9 cycle / sec

- Cycle period = $\frac{1}{2 \times 10^9 \text{ cycle/sec}} = \frac{1}{2} \times 10^{-9}$ sec / cycle

= 0.5 nanosec / cycle

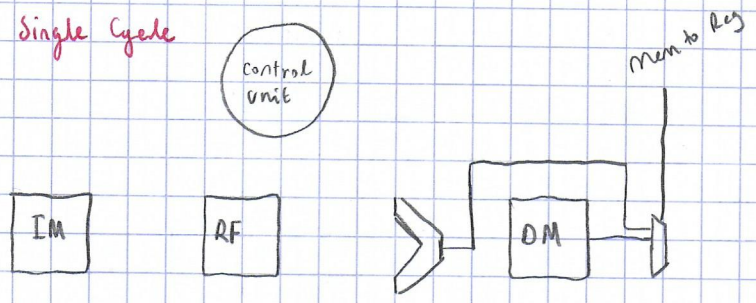
24 / 03 / 2017 , Thursday

add
sub
or
and
slt
} rd

sw
lw
} rt

beq

Single Cycle



Critical Path

IF
ID
Ex
M
WB

27 / 03 / 2017 , Monday

Interrupt / Exceptions

User Program

There is sth unusual
syscall
overflow
illegal instruction
incorrectly aligned data

Interrupt handling routine

Go back to the program.

actions to be performed:

1- Address of the instruction causing the exception / interrupt saved to EPC exception program counters.

2- Cause of the interrupt / exception => Cause register

3 - Exception Level
Saved to the status register

RTL

IM[PC]
{ There is a problem

Cause ← 40 // undefined inst.
EPC ← PC // instructions that follows the problem instruction
PC ← 0x 80 00 01 80

Multicycle Data Path

→ In single cycle, clock period is long enough to execute lw. Simple instruction require shorter amount of time

lw : IF, ID, EX, MEM, WB
sw : IF, ID, Ex, MEM, –
add : IF, ID, EX, –, WB

time.

$T_{lw} > T_{sw} > T_{add}$

→ For different no of instructions we have different no. of clock cycles. } Multi cycle.

→ Clock cycle should long enough to perform the longest Task.
↳ MEM.

→ Manifacturers use benchmark problems.

How to optimize the performance of a microprocessor for a certain problem domain

Program Profilers

Type a → arithmetic
Type b → logic
Type c → decision, brenching.

	Type A	Type B	Type C
Souree Prog	50	30	20
Execute Time	15	30	50

Example for exec. Time calculation in multicycle environments:

No. of instructions executed: 10^9
clock rate: 2 GHz

Type	CPI → clock cycle/ Instruction
A	3
B	4
C	5

Distribution of executed instructions among instruction types

50 % of type A
30 % of type B
20 % of type C

Final execution time

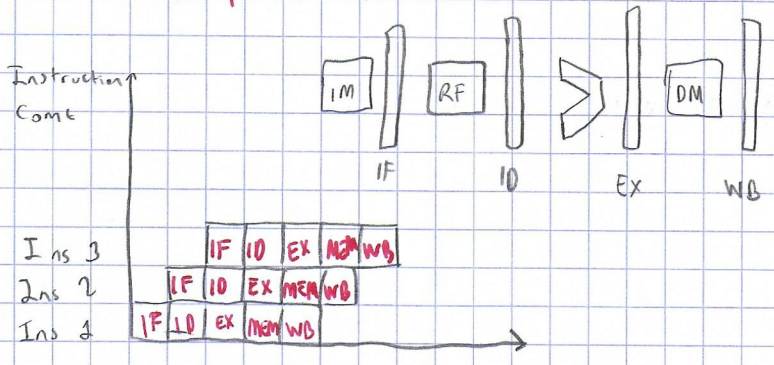
Ex Time = (Total number of clock cycles for the entire program) x (clock period)

$T_c = 10^9 \times (0.5 \times 3 + 0.3 \times 4 + 0.2 \times 5) = 10^9 \times 3.7$

$\text{Exec Time} = 10^9 \times 3.7 \times \frac{1}{2 \times 10^9 \text{ cycle/sec}} = 1.85 \text{ sec}$

Pipelining ← Aims to execute multiple ins. at the same time.

Parallel execution.
Instruction level
Program level.



Hazards

1- Data Hazard:

```
add $t1, $t2, $t3
and $t3, $t1, $t0
```

2- Control Hazard:

```
add $t1, $t2, $t3
beq $t5, $t6, Label
 —
 —
 —
Label1:
```

3- Structural Hazard: → The same unit may be needed by more than one instruction.

ALU

PC ← PC + 4

→ Consider the following timing requirements for different component

IM → $t_{mem}^{IF} = 250$ ns

$t_{RFread}^{ID} = 150$ ns

$t_{ALU}^{EX} = 200$ ns

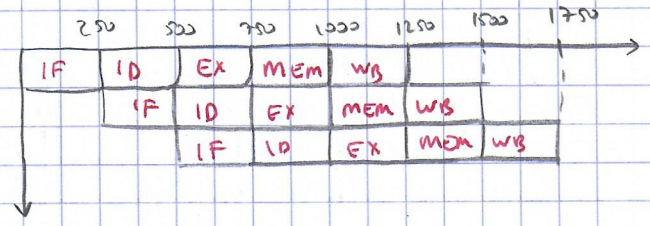
DM → $t_{mem}^{MEM} = 250$ ns

$t_{RFwrite}^{WB} = 100$ ns

+

950 ns

One ins takes: 950 ns



Consider executing 3 instructions

$t_{seq} = 3 \times 950 = 2850$ ns

$t_{pipeline} = 1750$ ns

Time needed to execute 10^9 inst.

$t_{seq} = (950 \times 10^{-9}) \times 10^9 = 950$ sec

$t_{pipeline} = (250 \times 10^{-9}) \times 10^9 = 250$ sec

Speedup ratio = $\frac{\text{time needed during sequential execution}}{\text{time needed during pipeline execution}}$

$= \frac{950}{250} \simeq 4$ times faster.

03/04/2017, Monday

Types of Dependencies

RAW: Read after write

```
add $1, $2, $3
add $5, $1, $1
```

Make sure that 1st instruction reads $2 before its written by 2nd instruction

WAR: Write after read

```
add $1, $2, $3
add $2, $0, $0
```

Can be a problem during concurrent execution of instructions.

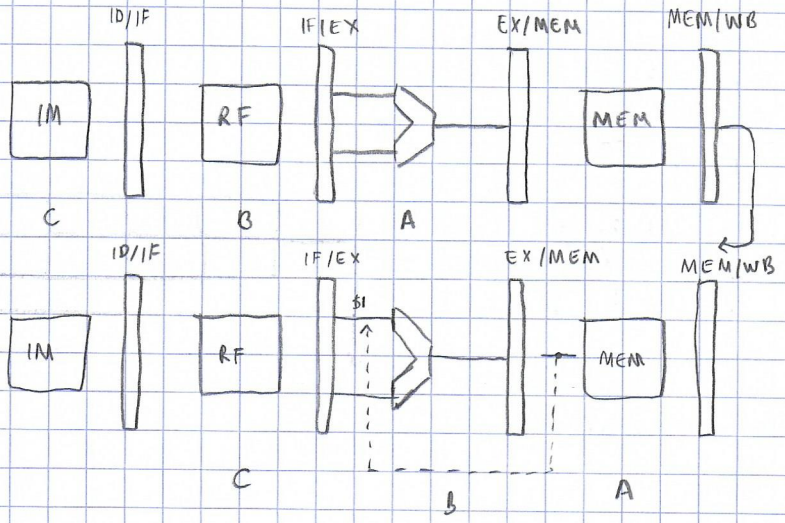
WAW: Write after write

```
add $1, $2, $3
add $1, $4, $5
```

Can be a problem during concurrent execution Make sure that instructions are executed in the order given/listed above

Data Dependencies:

A. add $1, $2, $3
B. add X, $1, X
C. add X, X, $1



EX/MEM.rd => ID/EX.rs

if (EX/MEM.rd = ID/EX.rs)

EX/MEM.rd => ID/EX.rs

06/04/2017, Thursday

Data Hazard Example:

- Solution with nop
- Assume no for wanding
  Find a sw solution
  by inserting nops.

```
lw  r4, 8(r12)
add r4, r4, r4
sw  r4, 8(r10)
```

stages: f/d/e/m/w

Clock c	1	2	3	4	5	6	7
lw	f	d	x	m	w		
add		f	d	x	m	w	

	1	2	3	4	5	6	7	8	9	10	11
lw	f	d	x	m	w						
nop		f	d	x	m	w					
nop			f	d	x	m	w				
add				f	d	x	m	w			
nop					f	d	x	m	w		
nop						f	d	x	m	w	
sw							f	d	x	m	w

Assume forwarding.

```
lw  r4, 8(r12)
add r4, r4, r4
sw  r4, 4(r10)
```

	1	2	3	4	5	6	7	8
lw	f	d	x	m	w			
nop		f	d	x	m	w		
add			f	d	x	m	w	
sw				f	d	x	m	w

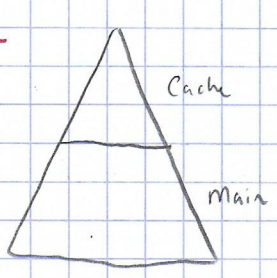
10/04/2017, Monday.

CHAPTER 8

Locality:

- → Space / Spatial Locality
- → Temporal Locality

E.x:



	1	2	3	4	1	3
M. memory	1 2 3 4 5 6	1 2 3 4 5 6	1 2 3 4 5 6	1 2 3 4 5 6	1 2 3 4 5 6	1 2 3 4 5 6
Cache	1	1 2	1 2 3	2 3 4	3 4 1	
	Miss	Miss	Miss	Miss	Miss	Hit

→ Assume that cache size = 3 data items

→ Assume, cache brings the items one by one

Hit Rate = 1/6

Miss Rate = 5/6

E.x:

	1	2	3	4	1	3
Cache	12	12	12	12	12	12
			34	34	34	34
Main Memory	12 34 56	123 456	123 456	123 456	123 456	123 456
	M	H	M	H	H	H

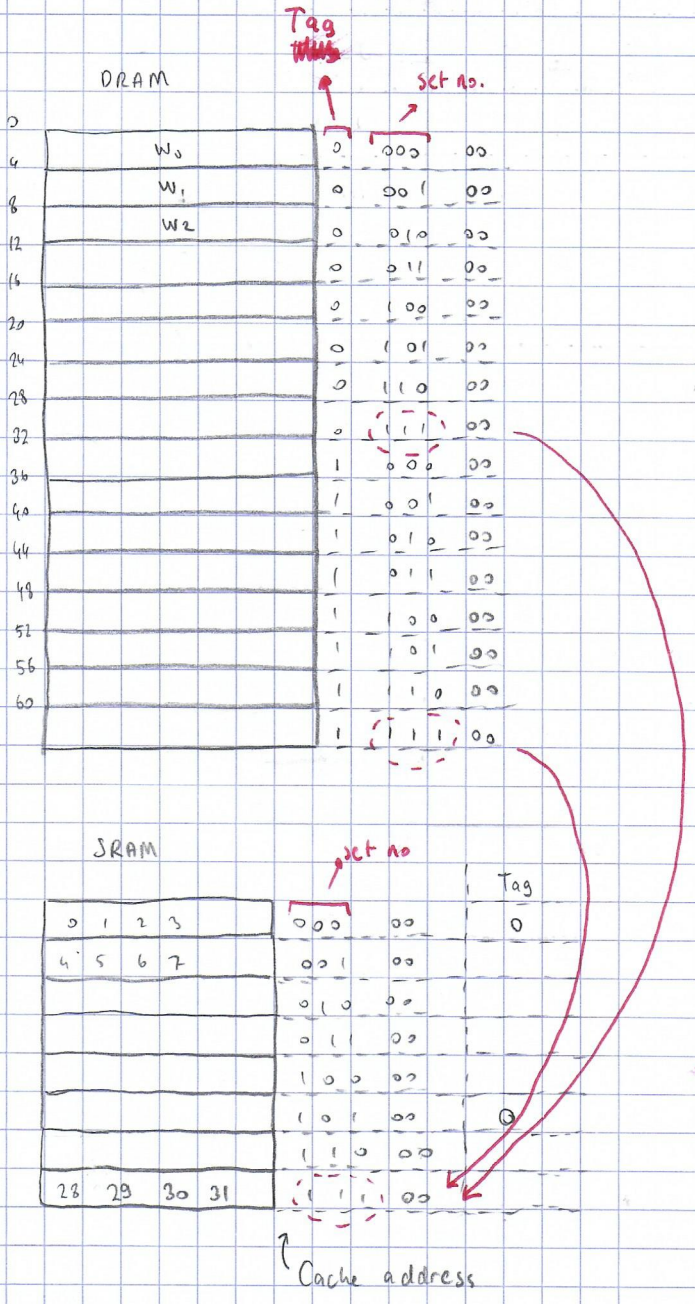
Cache size = 2x ? items => Taking 2 blocks that consists two items.

→ If it is available in the cache, we do not bring from main memory. That's why we missed in the third step.

Direct Mapped Cache

Memory size = 16 words = 64 bytes
Cache mem size = 8 words
Block size = 1 word

Dec	Hex	Bin
0	0	00 0000
4	4	00 0100
8	8	
12	C	
16	10	
20	14	
24	18	
28	1C	
32	20	10 0000
36	24	
40	28	10 1000
44	2C	
48	30	
52	34	
56	38	
60	3C	11 1100



Tag Set No.

X XXX XX

Memory Access	Tag	Set No.	Byte No	Miss / Hit
0 = 0 000 00	0	000	Byte 0 of the word	Miss
20 = 0 101 00	0	101	Byte 0 of the word.	Miss
21 = 0 101 01	0	101	Byte 1	Hit
32 = 1 000 00	1	000	Byte 0	Miss
33 = 1 000 01	1	000	Byte 1	Hit
60 = 1 111 00	1	111	Byte 0	Miss
61 = 1 111 01	1	111	Byte 1	Hit
62 = 1 111 10	1	111	Byte 2	Hit

The tag of set no. 0 of cache memory is 0.

→ Consider two words per block

X XX XXX

Tag Set No.

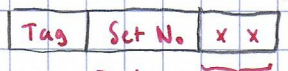
0	7	00	000
8	15	01	000
16	23	10	000
24	31	11	000

0	00	000	W_0	W_1	0
0	01	000	W_2	W_3	8
0	10	000			16
0	11	000			24
1	00	000			32
1	01	000			40
1	10	000			48
1	11	000	W_{14}	W_{15}	56
					64

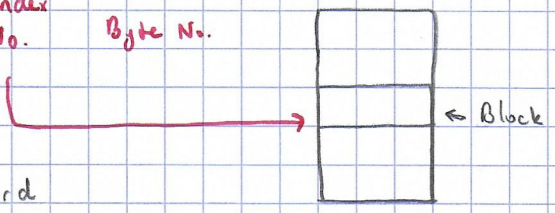
13/04/2019 Thursday

Direct Mapped Cache:

Physical address — Assume that each one is one word.



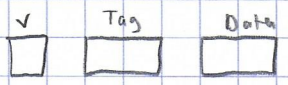
Index No. — Byte No.



Block = 2 word

X XX

Word No. Byte No.

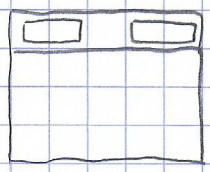


1 bit — Block

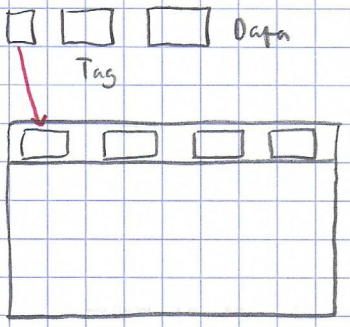
(V = Valid)

N-way associative memory

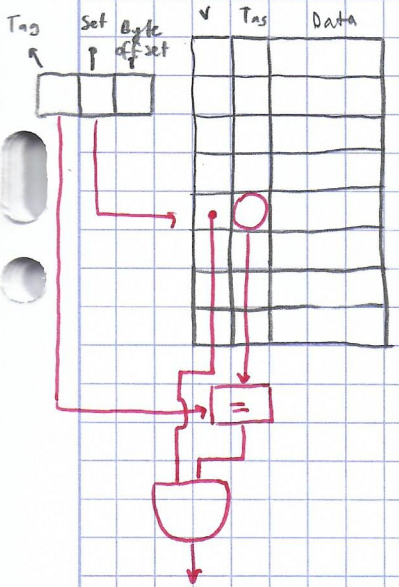
N = 2 (N = level of associativity)



N = 4



→ Assume 8 sets



Example: No. of sets 4K = $2^2 \times 2^{10} = 2^{12}$

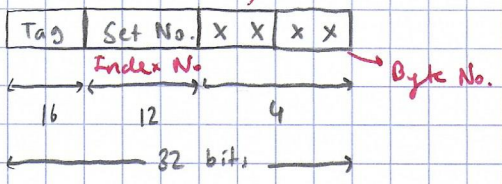
Block size = 4 words.

Physical Memory size = 4 GB = $2^2 \times 2^{30}$

Direct Mapped

Total No. of Tag bits

Physical Address



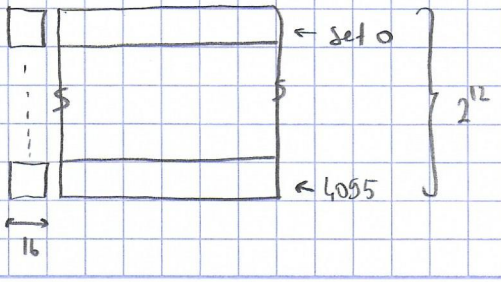
Total number of Tag bits =

No of sets x Tag size =

$2^{12} \times 2^4 = 2^{16}$ bits

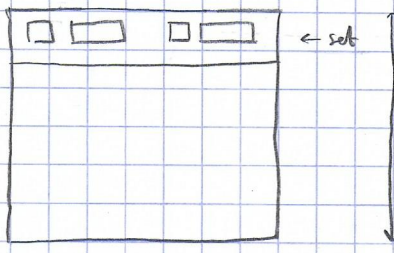
No. of sets — 16 tag bits

Cache Mem.

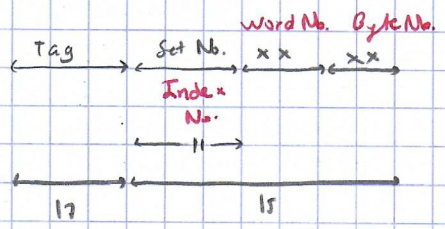


Cache Mem size = 4k x 4 words

→ n = 2



No. of Sets = 2k = $2 \times 2^{10} = 2^{11}$

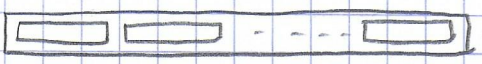


Block Size = 4 words

Total Number of Tag bits = $2^{11} \times 2 \times 17 =$

n = 2

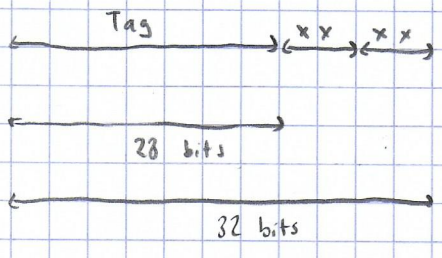
Full Associative:



No. of words = 4K x 4 words

= $2^{12} \times 2^2 = 2^{14}$ no. of words.

We have 4K blocks



Total number of Tag bits

= (No. of blocks) x (28)

= $\underbrace{2^2 \times 2^{10}}_{4K} \times 28$ bits

17/04/2017, Monday

First check cache
it's empty
cold start, miss

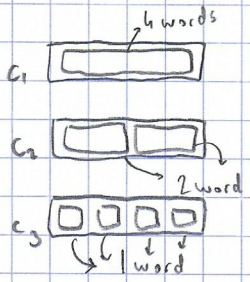
cache is full
item we want to access is not in cache
conflict.

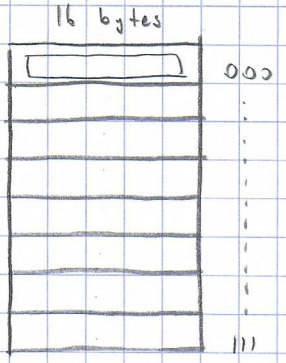
Example:

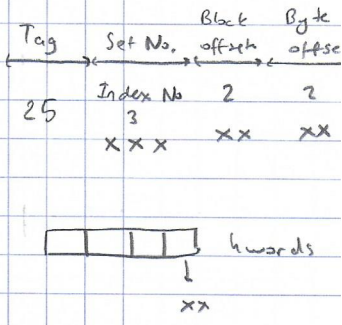
Cache mem size = 128 B
Phys. mem size = 4 GB = 32 bit

	N	Block size (words)
C_1	1	4 words
C_2	2	2 words
C_3	4	1 word

C_1: number of sets = $\frac{\text{Total cache mem size}}{\text{Set size}} = \frac{128}{4 \times 4} = \frac{2^7}{2^4} = 2^3$

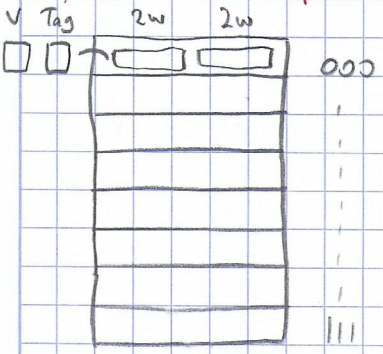


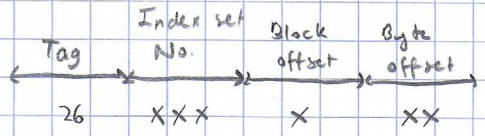




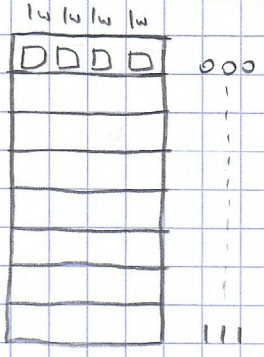
C_2: number of sets = $\frac{\text{Cache size}}{\text{set size}} = \frac{128}{N \times \text{Block size in words} \times \text{word size in bytes}} = \frac{2^7}{2 \times 2 \times 4} = 2^3$

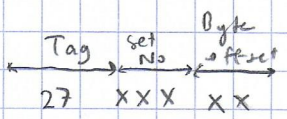
Depend on the tag, we will choose one of them.





C_3: number of sets = $\frac{\text{Cache size}}{N \times \text{Block size in words} \times \text{word size in bytes}} = \frac{128}{4 \times 1 \times 4} = 2^3$





C1 How to track byte position

Line No	V	T	11 Data 3	10 Data 2	01 Data 1	00 Data 0
000						
001						
≈						
111						

Consider accessing physical location 27

$27_{10} = 1B_{16} =$ 000 0011 1011

Set No — Block offset

Access byte number 4

C1

Line No	V	T	Data 1	Data 0
000				
001				
⋮	⋮	⋮	⋮	⋮
011	1	00 11	(223 - 220)	
≈				
111				

V	Tag	Data 1	Data 2
⋮	⋮	⋮	⋮
0			
≈			

$220_{10} = DC_{16} =$ 00 1101 1100

Tag — Set No. — Block offset

c)

Line	v	Tag	Data	v	Tag	Data	v	Tag	Data	v	Tag	Data
000												
≈												
111	1	0110	223-220	0			0			0		

↳ when we bring sth (occupied) we need to make it "1"

Access Mem loc 220

$220_{10} = DC_{16}$ - 0 . . . 000 1101 1100

20/04/2017, Thursday

Quiz 5 of Sec 686:

8 bit processor

byte oriented

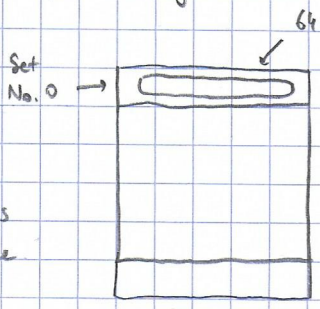
word = byte

64 MB memory

64 KB cache

Block size = 64 words

direct-mapped cache

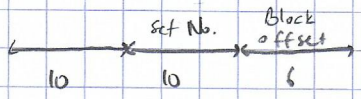


No of sets = $\frac{\text{Cache size}}{\text{Set size}}$

$= \frac{64 \times 2^{10}}{64} = 2^{10}$

Give address structure

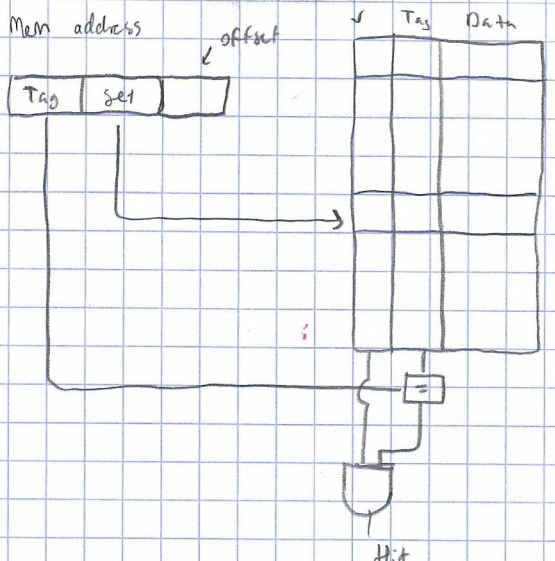
Memory address structure



$64\ MB = 2^6 \times 2^{20} = 2^{26}$

How many equality detectors

AND gates & OR gates

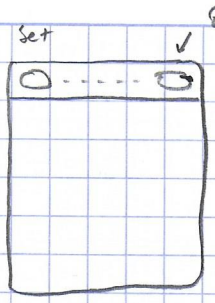


cold start

for 16 bit processor

each word
2 byte

64 MB memory
64 KB cache
Block size = 16 words
8 - way



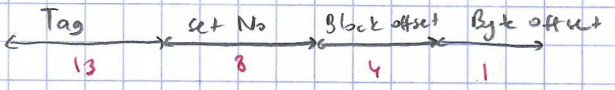
$$\text{No of sets} = \frac{\text{Cache size}}{\text{Set size}} = \frac{2^6 \times 2^{10}}{2^3 \times 2^4 \times 2^1}$$

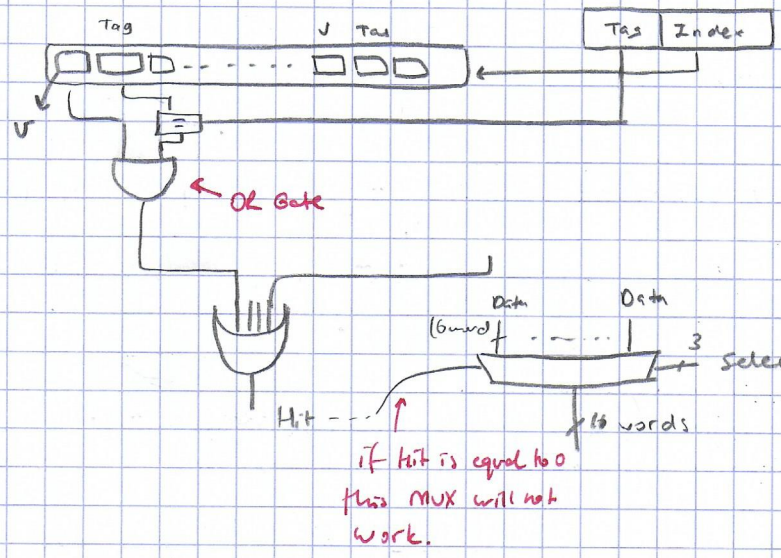
No of blocks/set

No of words/block

each word is 2 bytes.

Memory address format





Midterm Topics

- Single Cycle
  - RTL
  - Implement

Instruction → F D X M W

- Pipelining

→ Hazards → stall, forwarding, flush, nop

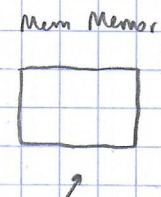
27/04/2017, Thursday

Virtual Memory

Each program has its own VM

Programs are seperated they have their own memory → better protection

Hard Disk
virtual memory
page



partitioned physical page.

Cache	VM
Block	Page
Block size	Page size
Block offset	Page offset
Miss	
Tag	Virtual page no.

Program is executed in terms of virtual addresses

Program uses virtual memory addresses

During execution, we have map to the physical memory addresses.

Virtual address → mapping → Physical address

we use

memory page table

Mapping is done during execution time

Dynamic address translation

like tag in caching

Virtual address: [Virtual page No. | Displacement offset within page]

Memory Paper table → OAT

VM >> PM

Physical address: [Physical Page No. | Offset]

example: VM = 8KB = $2^3 \times 2^{10} = 2^{13}$

Page Size = 1KB = 2^{10} = 1024

PM (Physical Memory) = 4KB = $2^2 \times 2^{10} = 2^{12}$

No of pages in VM = $\frac{2^{13}}{2^{10}} = 2^3 = 8$

No of pages (block) $\frac{2^{12}}{2^{10}} = 2^2 = 4$

VM

page 0	0	→ 000 00 0000 0000
page 1	1023 / 1024	→ 000 11 1111 1111 / 001 00 0000 0000
page 2	2048	→ 001 11 1111 1111 / 010 - - - -
page 3	3072	→ 011 - - - -
page 4	4096	→ 100 - - - -
page 5	5120	→ 101 - - - -
page 6	6144	→ 110 - - - -
page 7	7168	→ 111 - - - -

PM

Block 0	0	→ 00 00 0000 0000
Block 1	1024	→ 00 11 1111 1111 / 01 00 0000 0000
Block 2	2048	→ 01 11 1111 1111
Block 3	3072	→

Virtual Page No.

	V	Physical Page No.	Dist Address
000	0		
001	1	* 11	
010	1	00	
011	0		
100	0		
101	0		
110	0		
111	0		

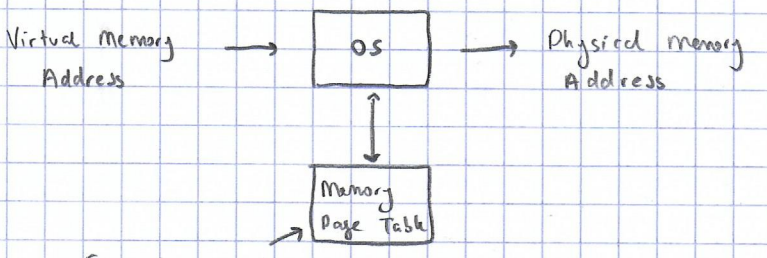
Virtual Address: [001 | 00 0000 0010]

Virtual page no. | offset

Memory Page Number

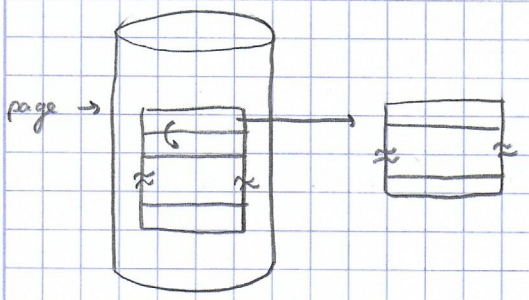
* [11 | 00 0000 0010]

Virtual Memory to Physical Memory Mapping

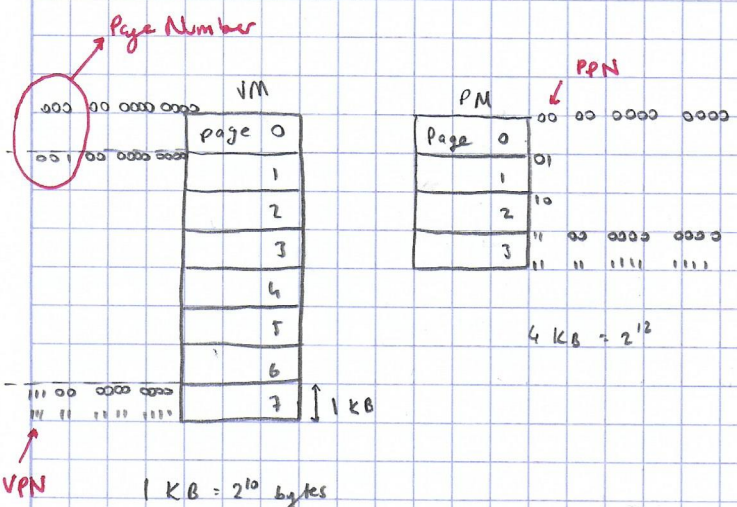


Each Program has its own memory page table.

- Memory Page table is kept in main memory.
- To make dynamic address translation some parts of a memory page table is kept in fast (cache-like) memory.
- TLB: Translation Lookaside Buffer.
- VM: Virtual Mem.
- PM: Physical mem.



- Virtual Memory Address refers to a virtual address whose page is not in main memory.



- VPN: Virtual Page Number
- PPN: Physical Page Number

1 KB = 2^{10} bytes

VM size = 8 KB => $2^3 \times 2^{10} = 2^{13}$

Memory Page Table

No. of rows = No of pages in VM

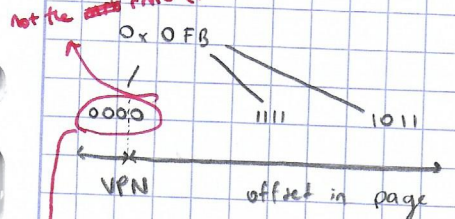
Page No.	Physical Page No.	V	D
000	10	1	1
001		0	-
010		0	-
011		0	-
100	11	1	0
101		0	-
110		0	-
111		0	-

- V = Valid bit => 1 means that virtual page is in main memory
- D = Dirty bit => 1 means that physical has been changed while it was in main memory

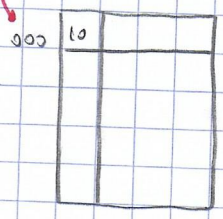
Table size for the memory page table

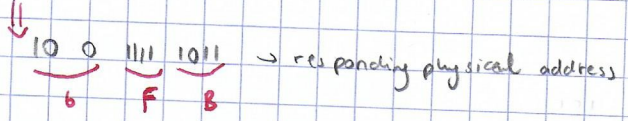
8 x (2 + 1 + 1) = 80 bits

Virtual Memory Address



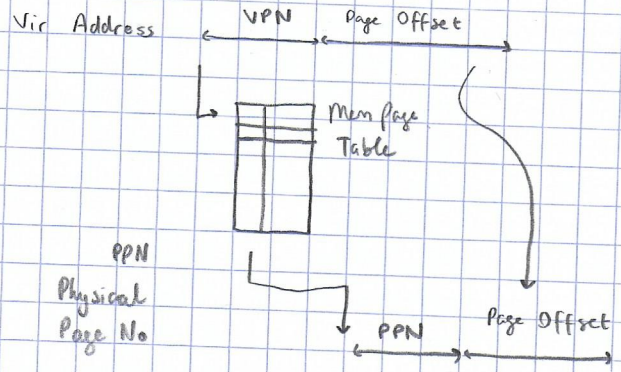
Mem. Page Table





→ responding physical address

DAT (Dynamic Address Translation)



8/05/2017

E.x: (~~B~~ from the textbook) C to MIPS conversion in PIC32. Quiz #6 (Spring 2015)

↳ 8.18

```
#include <P32xxxx.h>
int main(void)
{
    int switches;
    TRISD = 0xFF00;
    while(1)
    {
        switches (PORTD >> 8) && 0xF;
        PORTD = switches;
    }
}
```

sw

lw

$t1

TRISD = 0xBF 88 60 C0

PORTD = 0xBF 88 60 D0

$t2

Solution:

```
main:  # $t1 = TRISD , $t2 = PORTD
lui $t0, 0xBF88
ori $t1, $t0, 0x60C0
ori $t2, $t0, 0x60D0
addi $t3, $0, 0xFF00   → storing { # TRISD = 0xFF0
sw $t3, 0($t1)
while: bne $0, $0, end   → We do not need to write it
lw $t4, 0($t2)
srl $t4, $t4, 8
andi $t4, $t4, 0xF   # $t4 = switches
sw $t4, 0($t2)
jump while
end:
```

While loop

Example: # Q1 (Spring 2014 Final)

```
void routine2 (void)
{
   int temp;
   TRISB = 0x000F;
   while ((PORTB & 0xF) != 0x0)
   {
      temp = PORTB;
      PORTB = temp << 4;
   }
}
```

TRISB = 0xFF CC 00 00 ← $t0

PORTB = 0xFF CC 00 04 ← 4($t0)

```
routine 2:
lui $t0, 0xFFCC
ori $t0, $t0, 00 00
addi $t1, $0, 0x 00 0F      } Before while
sw $t1, 0($t0)
while:
lw $t1, 4($t0)        #Read PortB
andi $t2, $t1, 0x F
beq $t2, $0, exit
# Read from PortB     } We don't need this part. We did it above.
lw $t1, 4($t1)
sll $t2, $t1, 4
sw $t2, 4($t0)
jump while
exit:
jr $ra
```

While loop

Example: # Q1 (Spring 2014 Retake)

```
int problem_1b( int a) ← comes in $a0
{
   TRISD = 0x FF00;
   PORTD = a;
   while((PORTD & 0xFF00) == 0)
   {
      return PORTD; → returned in $v0
   }
}
```

PORTD = 0xFF FF 00 00 ← $t0

TRISD = 0xFF FF 00 04 ← 4($t0)

$t0 = PORTD

my solution

```
problem_1b:
   lui $t0, 0xFF FF
   ori $t0, 0x00 00
   addi 4($t0), $0, 0xFF00
   addi $t0, $t0, $a0
while:
   andi $t1, 0($t0), 0xFF00
   bne $t1, $0, end
   li $v0, 0($t0)

end:
   jr $ra
```

Answer:

```
problem_1b:
   lui $t0, 0xFFFF
   addi $t1, $0, 0xFF 00
   sw $t1, 4($t0)
   sw $a0, 0($t0)
spin:
   lw $t2, 0($t0)
   andi $t3, $t2, $t1
   beq $t3, 0, spin
   lw $v0, 0($t0)
   jr $ra
```

11/05/2017, Thursday

Question #1 (Spring 2015 Final)

Example: AD1CON1bits.SAMP = 0;

```
lui  $t0, 0xBF80;
ori  $t0, $t0, 0x9000
lw   $t1, 0($t0) //Loading the content of the register.
andi $t1, $t1, 0xFFFD
sw   $t1, 0($t0)
```

Question #1 (Spring 2015 Final):

```
int readadc(void)
{
    AD1CON1bits.SAMP = 0;
    while(!AD1CON1bits.DONE);
    AD1CON1bits.SAMP = 1;
    AD1CON1bits.DONE = 0;
    return (AD1BUF0);
}
```

```
#while(!AD1CON1bits.DONE)
while: lw   $t1, 0($t0)
       andi $t2, $t1, 0x0001
       beq  $t2, $0, while

#AD1CON1bits.SAMP = 1;
       ori  $t1, $t1, 0x0002
#AD1CON1bits.DONE = 0;
       andi $t1, $t1, 0xFFFE
       sw   $t1, 0($t0)

#return (AD1BUF0);
       lw   $v0, 0x9070($t0)
       jr   $ra
```