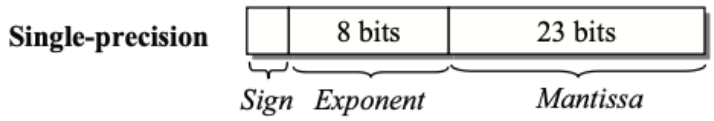
IEEE Floating Point representation

- 32 bits. Exponent is in excess 127



Example:

Convert hexadecimal C2060000 represented in the IEEE Standard 754 single-precision floating point representation to the decimal value it represents.

$(C2060000)_{16} = (1100\ 0010\ 0000\ 0110\ 0000\ 0000\ 0000\ 0000)_2$

1	10000100	00001100000000000000000

Exponent = 132 – 127 = 5

Value = $-(1.000011)_2 \times 2^5 = -(100001.1)_2 = \mathbf{-(33.5)_{10}}$

Complements:

(R-1)'s complement (Given n-digit base-R number X):

- $-X = R^n - X - 1$
- $(-22)_{10} = 10^2 - 22 - 1 = (77)_{9s}$

R's complement:

- $-X = R^n - X$
- $(-22)_{10} = 10^2 - 22 = (78)_{10s}$

1's complement (Range: $[-2^{n-1}-1$ to $2^{n-1}-1]$):

- $-X = 2^n - X - 1$
- Convert from negative binary just flip the bits
- Addition is binary addition but needs to add carry-out

2's complement (Range: $[-2^{n-1}$ to $2^{n-1}-1]$):

- $-X = 2^n - X$
- Convert from negative binary just flip the bits and add 1
- Addition discards carry-out

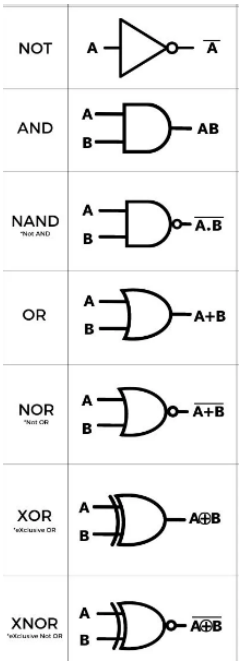
Overflow – Occurs if carry-in and carry-out of MSB are different or if A and B have same sign but result has opposite sign

BCD addition – Add 0110(6) to invalid portions after performing binary addition

Pass by value – In C all arguments are passed by value, including structs. Array names are fixed pointers(cannot be reassigned)

Logical Operators (Bitwise)

Operators	Effect	C
AND	-	&
OR	-	|
NAND	A NAND B = A’ + B’	-
NOR	A NOR B = A’.B’	-
XOR	A XOR B = A.B’ + A’.B	^
XNOR	A XNOR B = A.B + A’.B’	-
NOT	Invert bits	~

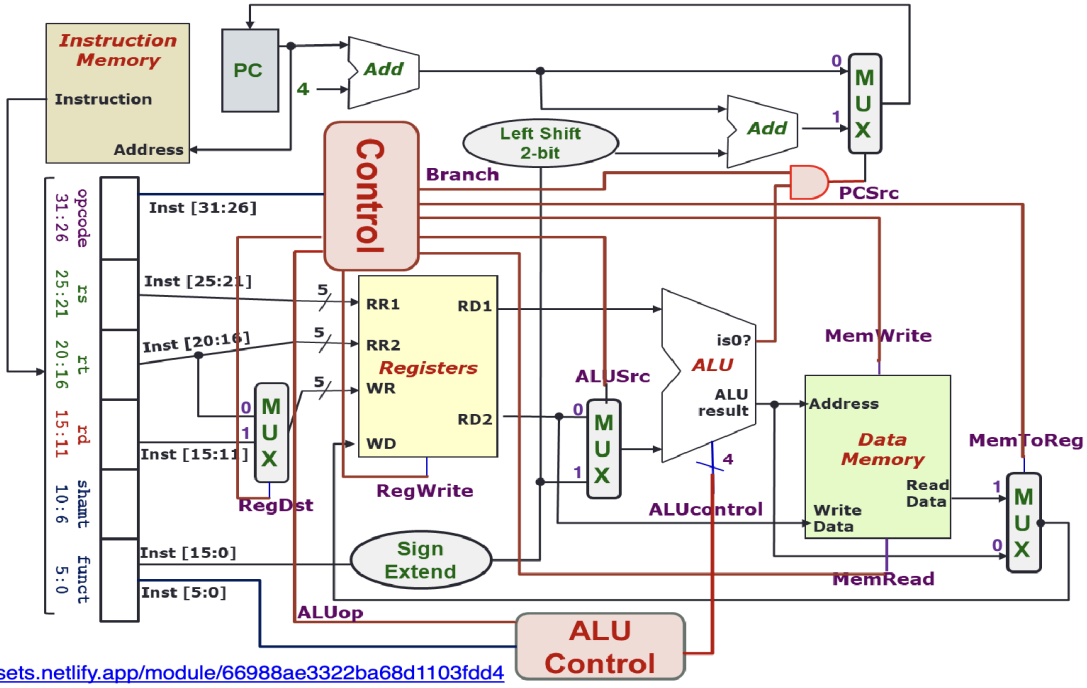


Opcode formula(A < B < C):

Max – $2^{C} - 2^{C-B} + 1 - 2^{C-A} + 1$

Min – $2^{A} - 1 + 2^{B-A} - 1 + 2^{C-B}$

Datapath/Control path



//sets.netlify.app/module/66988ae3322ba68d1103fdd4

- MemToReg mux is inverted (So the lines doesn’t crossover in the diagram)

Control signals

Signal	Stage	Purpose	Values
`RegDst`	Decode	Select the destination register number	• `0`: `$rt` • `1`: `$rd`
`RegWrite`	Decode/Writeback	Enable writing of register	• `0`: no write • `1`: write
`ALUSrc`	ALU	Select the 2nd operand for ALU	• `0`: `$rt` • `1`: `immediate`
`ALUcontrol`	ALU	Select the operation to be performed by the ALU	see `ALUcontrol` table
`MemRead`	Memory	Enable reading of data memory	• `0`: no read • `1`: read[1]
`MemWrite`	Memory	Enable writing of data memory	• `0`: no write • `1`: write[1]
`MemToReg`	Writeback	Select the result to be written back to register file	• `0`: ALU result • `1`: memory data[2]
`Branch`[3]	Memory/Writeback	Select the next `$PC` value	• `0`: `$PC+4` • `1`: `($PC+4)+(immediate×4)`

ALUcontrol	Function
0000	`and`
0001	`or`
0010	`add`
0110	`sub`
0111	`slt`
1100	`nor`

Endianness – Relative ordering of bytes in a multi-byte word stored in memory. Big endian → most significant byte stored in lowest address

MIPS assembly language

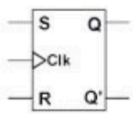
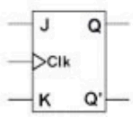
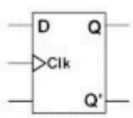
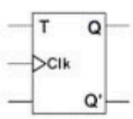
Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Used to add constants
Data transfer	load word	`lw $s1, 100($s2)`	`$s1 = Memory[$s2 + 100]`	Word from memory to register
	store word	`sw $s1, 100($s2)`	`Memory[$s2 + 100] = $s1`	Word from register to memory
	load byte	`lb $s1, 100($s2)`	`$s1 = Memory[$s2 + 100]`	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	`Memory[$s2 + 100] = $s1`	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	`beq $s1, $s2, 25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1, $s2, 25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1, $s2, $s3`	if (`$s2 < $s3`) `$s1` = 1; else `$s1` = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if (`$s2 < 100`) `$s1` = 1; else `$s1` = 0	Compare less than constant
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

Flip Flops

1. Flip-flops are synchronous bistable devices.
2. Change state either at the positive (rising) edge, or at the negative (falling) edge of the clock signal. (Basically, edge triggered)
3. Preset(PRE) and Clear(CLR) are asynchronous input to flip flop
    a. PRE active, Q immediately high.
    b. CLR active, Q immediately low
4. Latches are pulse triggered(The horizontal region), otherwise similar to flip flips

Solving flip flops

1. Draw truth tables with A,B and A^+,B^+ and flip flop inputs

FLIP-FLOP NAME	FLIP-FLOP SYMBOL	CHARACTERISTIC TABLE
SR		S R Q(next) 0 0 Q 0 1 0 1 0 1 1 1 ?
JK		J K Q(next) 0 0 Q 0 1 0 1 0 1 1 1 Q'
D		D Q(next) 0 0 1 1
T		T Q(next) 0 Q 1 Q'

- *Excitation tables*: given the required transition from present state to next state, determine the flip-flop input(s).

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

JK Flip-flop

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

SR Flip-flop

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

D Flip-flop

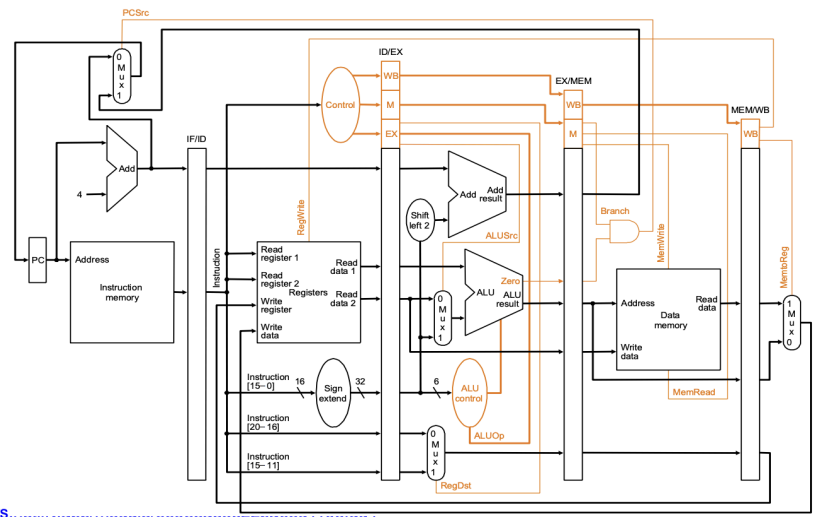
Q	Q^+	T
0	0	0
0	1	1
1	0	1
1	1	0

T Flip-flop

Pipeline stages

1. IF: Instruction Fetch
2. ID: Instruction Decode and Register Read
3. EX: Execute an operation or calculate an address
4. MEM: Access an operand in data memory
5. WB: Write back the result into a register

4. Pipeline Control: **Datapath** and **Control**



https://sets

Control signals

	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

Pipeline hazards

1. Structural hazards – Simultaneous use of a hardware resource
2. Data hazards – Data dependencies between instructions
3. Control hazards – Change in programme flow

Processor performance

Single cycle – Time = I x CT(Cycle time)

- I – no. of instructions
- Cycle time – longest instruction

Multi cycle – Time = I x Average Cycles/Instruction x CT

- Cycle time – longest stage

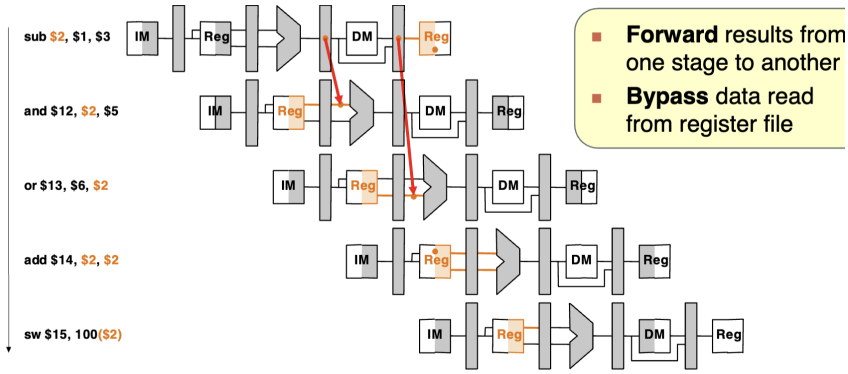
Pipeline – Time = (I + N - 1) x CT

- N – no. of stages
- Cycle time – longest stage + overhead(eg. Pipeline register latency)

Forwarding

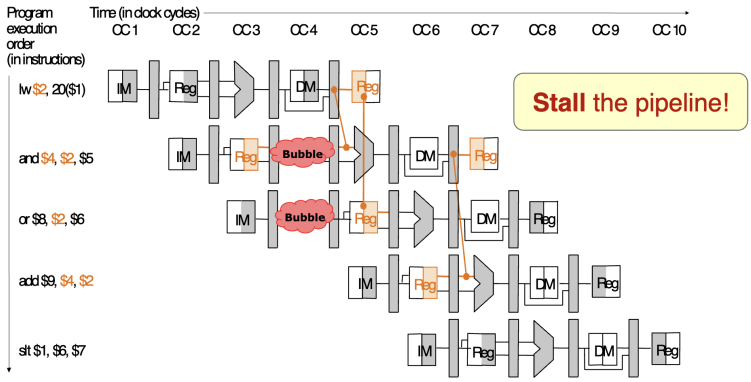
Add

1. Can direct from ALU out to ALU in



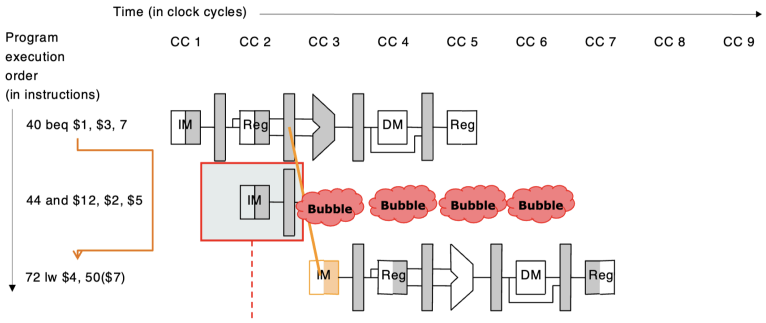
Load

1. Can from DM out to ALU in. Usually +1 if before R-type



Branching

1. Early branching
    a. Move branch to ID stage (+2 if lw before, +1 if R-type before)
2. Branch prediction
    a. Assume branch not taken
    b. Correct prediction, no stall
    c. Wrong prediction need to flush(+1)



3. Move non-control dependent instructions to slot behind branch
    a. Usually instruction is before the branch
    b. Can usually find 50% of time

No enhancement

1. R-Type: +2
2. lw: +2
3. Branch: +3
4. Generally delay ID to previous WB (Applicable to most situations)

Mem hierarchy

Register > SRAM(cache) > DRAM(main memory) > hard drive

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

1. Miss time – Hit time + time needed to replace cache block

(Valid[index] = TRUE) AND (Tag[index] = Tag[memory address]) → cache hit

Cache misses

1. Compulsory misses
    a. On the first access to a block; the block must be brought into the cache
    b. Also called cold start misses or first reference misses
2. Conflict misses (Only for direct map and SA cache)
    a. when several blocks are mapped to the same block/set
    b. Also called collision misses or interference misses
3. Capacity misses (Only for FA cache)
    a. Occur when blocks are discarded from cache as cache cannot contain all blocks needed

Write Policy

1. Write-through cache
    a. Write data both to cache and to main memory
    b. With buffer, like a printer queue
2. Write-back cache
    a. Mainly write to cache, write to main memory only when cache block is replaced
    b. Add a dirty bit, if write, change dirty bit to 1. When change cache, write back if dirty bit is 1

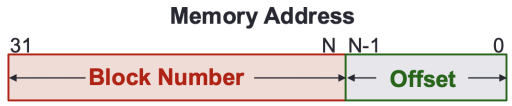
Write miss solutions

1. Write allocate – Load block into cache, leave memory to write policy
2. Write around – Write to memory only, don't touch cache

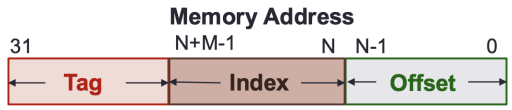
Block replacement policy (For SA/FA cache)

1. Least Recently Used
2. First in first out
3. Random replacement
4. Least frequently used

Direct mapped cache



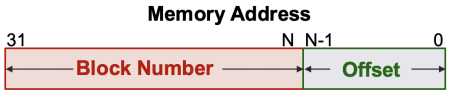
Cache Block size = 2^N bytes



Cache Block size = 2^N bytes
Number of cache blocks = 2^M
Offset = **N bits**
Index = **M bits**
Tag = **32 – (N + M) bits**

1. Total size not necessarily 32. Eg.16 GB=16×2^{30} bytes= 2^{34} bytes → size = 34
2. Block number = total - offset

Set associative cache (SA)



Cache Block size = 2^N bytes

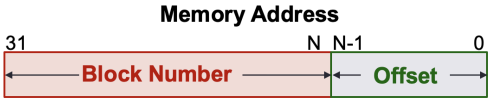
Cache Set Index
= (BlockNumber) modulo (NumberOfCacheSets)

Cache Block size = 2^N bytes
Number of cache sets = 2^M
Offset = **N bits**
Set Index = **M bits**
Tag = **32 – (N + M) bits**

Observat
It is esse
unchange
direct-ma

1. Sets of blocks instead of 1 single block per index

Fully associative cache (FA)



Cache Block size = 2^N bytes

Cache Block size = 2^N bytes
Number of cache blocks = 2^M
Offset = **N bits**
Tag = **32 – N bits**

Obse
The
serve