COD Review

-- SugarHe

0. Basic

- 1 byte = 8 bits
- 1 word = 4 bytes = 32 bits
- 1 doubleword = 8 bytes = 64 bits

1. RISC-V Assembly

1.1 operation

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	`add x5, x6, x7`	`x5 = x6 + x7`	Three register operands; add
	Subtract	`sub x5, x6, x7`	`x5 = x6 - x7`	Three register operands; subtract
	Add immediate	`addi x5, x6, 20`	`x5 = x6 + 20`	Used to add constants
Data transfer	Load doubleword	`ld x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Doubleword from memory to register
	Store doubleword	`sd x5, 40(x6)`	`Memory[x6 + 40] = x5`	Doubleword from register to memory
	Load word	`lw x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Word from memory to register
	Load word, unsigned	`lwu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned word from memory to register
	Store word	`sw x5, 40(x6)`	`Memory[x6 + 40] = x5`	Word from register to memory
	Load halfword	`lh x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Halfword from memory to register
	Load halfword, unsigned	`lhu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned halfword from memory to register
	Store halfword	`sh x5, 40(x6)`	`Memory[x6 + 40] = x5`	Halfword from register to memory
	Load byte	`lb x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Byte from memory to register
	Load byte, unsigned	`lbu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Byte unsigned from memory to register
	Store byte	`sb x5, 40(x6)`	`Memory[x6 + 40] = x5`	Byte from register to memory
	Load reserved	`lr.d x5, (x6)`	`x5 = Memory[x6]`	Load; 1st half of atomic swap
	Store conditional	`sc.d x7, x5, (x6)`	`Memory[x6] = x5; x7 = 0/1`	Store; 2nd half of atomic swap
	Load upper immediate	`lui x5, 0x12345`	`x5 = 0x12345000`	Loads 20-bit constant shifted left 12 bits
Logical	And	`and x5, x6, x7`	`x5 = x6 & x7`	Three reg. operands; bit-by-bit AND
	Inclusive or	`or x5, x6, x8`	`x5 = x6 | x8`	Three reg. operands; bit-by-bit OR
	Exclusive or	`xor x5, x6, x9`	`x5 = x6 ^ x9`	Three reg. operands; bit-by-bit XOR
	And immediate	`andi x5, x6, 20`	`x5 = x6 & 20`	Bit-by-bit AND reg. with constant
	Inclusive or immediate	`ori x5, x6, 20`	`x5 = x6 | 20`	Bit-by-bit OR reg. with constant
	Exclusive or immediate	`xori x5, x6, 20`	`x5 = x6 ^ 20`	Bit-by-bit XOR reg. with constant
Shift	Shift left logical	`sll x5, x6, x7`	`x5 = x6 << x7`	Shift left by register
	Shift right logical	`srl x5, x6, x7`	`x5 = x6 >> x7`	Shift right by register
	Shift right arithmetic	`sra x5, x6, x7`	`x5 = x6 >> x7`	Arithmetic shift right by register
	Shift left logical immediate	`slli x5, x6, 3`	`x5 = x6 << 3`	Shift left by immediate
	Shift right logical immediate	`srli x5, x6, 3`	`x5 = x6 >> 3`	Shift right by immediate
	Shift right arithmetic immediate	`srai x5, x6, 3`	`x5 = x6 >> 3`	Arithmetic shift right by immediate
Conditional branch	Branch if equal	`beq x5, x6, 100`	`if (x5 == x6) go to PC+100`	PC-relative branch if registers equal
	Branch if not equal	`bne x5, x6, 100`	`if (x5 != x6) go to PC+100`	PC-relative branch if registers not equal
	Branch if less than	`blt x5, x6, 100`	`if (x5 < x6) go to PC+100`	PC-relative branch if registers less
	Branch if greater or equal	`bge x5, x6, 100`	`if (x5 >= x6) go to PC+100`	PC-relative branch if registers greater or equal
	Branch if less, unsigned	`bltu x5, x6, 100`	`if (x5 < x6) go to PC+100`	PC-relative branch if registers less, unsigned
	Branch if greater or equal, unsigned	`bgeu x5, x6, 100`	`if (x5 >= x6) go to PC+100`	PC-relative branch if registers greater or equal, unsigned
Unconditional branch	Jump and link	`jal x1, 100`	`x1 = PC+4; go to PC+100`	PC-relative procedure call
	Jump and link register	`jalr x1, 100(x5)`	`x1 = PC+4; go to x5+100`	Procedure return; indirect call

1.2 operands

asm var are registers

- operands are registers
- number is fixed
- fast
- numbered from 0 to 31
- x0 is always 0, can be used for assignment

1.2.1 Add & Sub

for integers, has associativity

addi use signed imm, thus there is no subi

imm will be filled with sign bit to the same length

1.2.2 Load & Store

from A[3] to **x10**

lw x10, 12(x15)

- 12 is the offset (in bytes, multiplication of 4)
- **x15** is the base register (A[0])

from **x10** to A[3]

sw x10, 12(x15)

byte data transfer: **lb**, **sb**

lb need to extend sign bit (**sign-extend**)

store does not change other bytes

1.2.3 Decision

- conditional branch: **beq** branck if equal, **bne** branch if not equal
- unconditional branch: **j**

blt rs1, rs2, label

branch when rs1 < rs2, use signed integers comparison

check out bounds, use unsigned comparison, negative number is always larger than positive number

bgeu rs1, rs2, OutofBound

compare **rs1** with **rs2** to tell whether **rs1** $\in [0, \mathbf{rs2}]$

1.2.4 Logical

- **andi** can be used for mask
- **not** is replaced by **xor 11111111**
- **sll**, **srl** shift and insert 0
- **sra** shift and insert sign bit

- left shift n bits is multiplying with 2^n, right shift n bits for positive number is dividing by 2^n, but not for negative number

1.3 Calling Function

1. put parameters in certain reg
2. transfer control to callee
3. acquire local resource
4. perform function task
5. store result in certain reg
6. return control to the caller

jal x1, label

jump and link, store the next pc to x1

$PC = PC + imm$

jalr x0, 0(x1)

jump back, (use **jr** as pseudo inst)

use stack to store the old values in regs

sp is stack pointer in **x2**, started from high to low

- push, decrease **sp**, and **sw**
- pop, **lw**, and increase **sp**

use stack to save certain regs

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

1.3.2 Memory

- static: var declared once per program, global or constant, **gp**
- text segment: store the machine code
- stack: used during execution, save reg
- heap: save var dynamically declared by $malloc$

2 RISC-V ISA

RISC-V inst are 32 bits, comprised of different fields

- **opcode**: identify inst type
- **rs2**: source reg 2, always before **rs1**
- **rs1**: source reg 1
- **rd**: destination reg
- **funct7+funct3**: combine with **opcode** to describe the specific operation
- **imm**" immediate number, at most 20 bits divided into different parts

2.1 R-format

reg-reg arithmetic operation

0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub
0000000	rs2	rs1	001	rd	0110011	sll
0000000	rs2	rs1	010	rd	0110011	slt
0000000	rs2	rs1	011	rd	0110011	sltu
0000000	rs2	rs1	100	rd	0110011	xor
0000000	rs2	rs1	101	rd	0110011	srl
0100000	rs2	rs1	101	rd	0110011	sra
0000000	rs2	rs1	110	rd	0110011	or
0000000	rs2	rs1	111	rd	0110011	and

Different encoding in funct7 + funct3 selects different operations

2.2 I-format

reg-imm arithmetic operation & load

- 12 bits imm, signed, sign-extended to 32 bits
- **srli, slli**, only use 5 bits imm (32 bits reg), the lower 5 bits
- load inst use **funct3** to specify size and sign (sign not necessary for a word)

imm[11:0]		rs1	000	rd	0010011	addi
imm[11:0]		rs1	010	rd	0010011	slti
imm[11:0]		rs1	011	rd	0010011	sltiu
imm[11:0]		rs1	100	rd	0010011	xori
imm[11:0]		rs1	110	rd	0010011	ori
imm[11:0]		rs1	111	rd	0010011	andi
0000000	shamt	rs1	001	rd	0010011	slli
0000000	shamt	rs1	101	rd	0010011	srli
0100000	shamt	rs1	101	rd	0010011	srai

imm[11:0]	rs1	000	rd	0000011	lb
imm[11:0]	rs1	010	rd	0000011	lh
imm[11:0]	rs1	011	rd	0000011	lw
imm[11:0]	rs1	100	rd	0000011	lbu
imm[11:0]	rs1	110	rd	0000011	lhu

2.3 S-format

store

replace **rd** with lower 5 bits of **imm**

mem[rs1 + imm] = rs2

Imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	sb
Imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	sh
Imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	sw

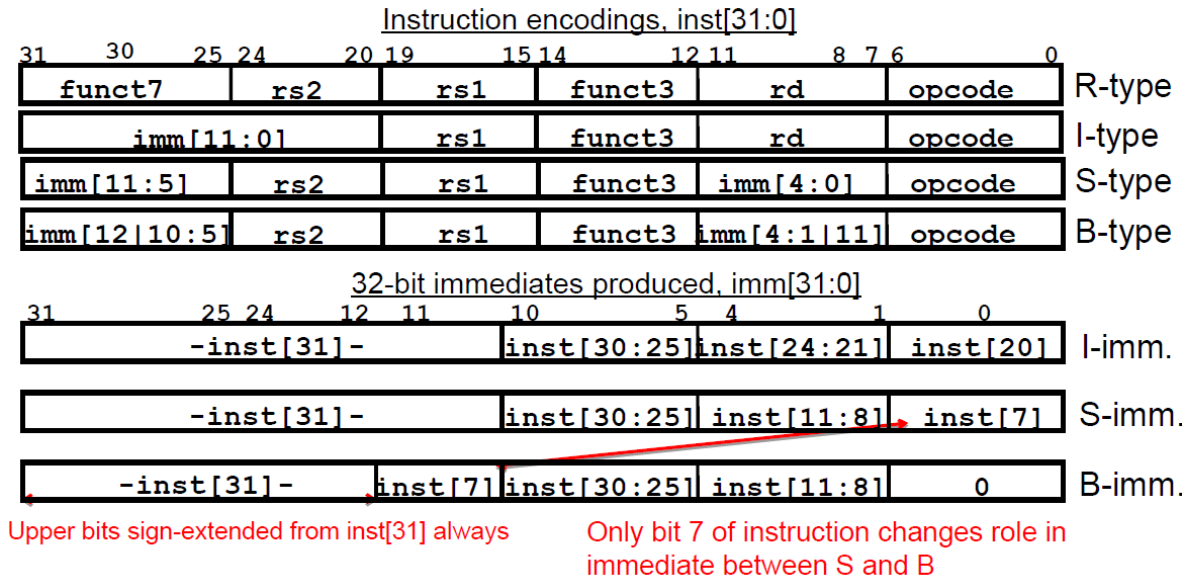
2.4 B-format

branch, a variant of S-format, called SB-format before

PC-Relative-Addr use **imm** as PC offset, thus specify $\pm 2^{11}$ unit, use 2 bytes as a unit (16 bits system), thus $\pm 2^{10}$ inst, $\pm 2^{12}$ bytes

			funct3		opcode	
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU

keeps imm construction similar to S-imm & I-imm, only need to move imm[11]

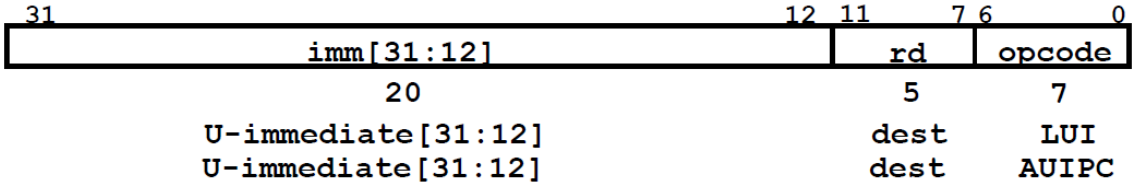


2.5 U-format

support for 20 bits imm

lui load imm to reg, **auipc** add imm to pc & store to reg

- create a 32 bits imm, write the upper 20 bits & clear lower 12 bits
- use **addi** to set lower 12 bits **why not ori?** (**ori** will use sign-extend too, it is wrong)
- **addi** is sign-extended to 32 bits, if sign bit of 12 bits is 1, need to add 1 to upper 20 bits (clear all the sign extension)



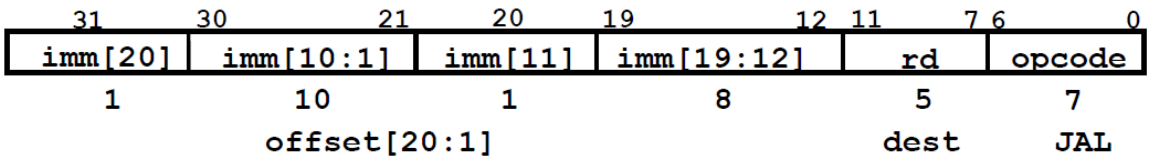
2.6 J-format

for jumps, a variant of U-format, called UJ-format before

- have a 20 bits imm, specify $\pm 2^{19}$ units
- use 2 bytes as a unit, $\pm 2^{18}$ inst, $\pm 2^{20}$ bytes

Note

- **jalr** is I-format
- write $PC + 4$ to **rd** (x1)
- set $PC = rs + imm$
- **imm** is not unit of 2 bytes



3 Performance Metrics

3.1 Execution Time

- Elapse Time: disk & mem access, I/O
- CPU Time: time spent by a program in CPU, including system & user
- User CPU Time: **focus** (Execution Time)

3.2 Clock Cycles

- clock rate: cycles per sc
- $\text{Execution Time} = \text{Clock Cyles} \times \text{Clock Cycle Time} = \frac{\text{Clock Cyles}}{\text{Clock Rate}}$
- $\text{Clock Cycles} = \#\text{Inst} \times CPI$

final measurement

$$\text{Execution Time} = \frac{\#\text{Inst} \times CPI}{\text{Clock Rate}}$$

- inc clock rate
- dec CPI
- reduce # Inst

3.3 Amdahl's Law

$\text{Speed up} = \frac{\text{old Execution Time}}{\text{new Execution Time}}$

if improve a fraction f by factor a

$$\text{Speed up} = \frac{1}{(1-f) + \frac{f}{a}}$$

- at most $\frac{1}{1-f}$

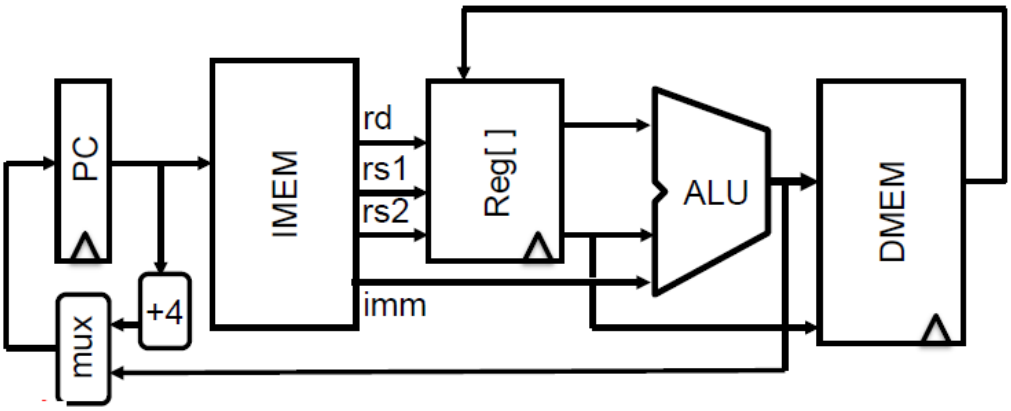
4 Datapath

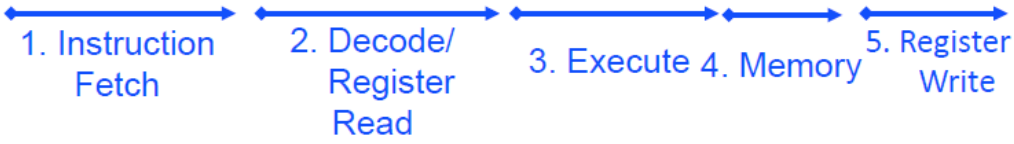
CPU is comprised of 3 parts

- Datapath: perform operation
- Control: tell datapath what to do
- Memory: store data & inst
- I/O: input & output

Basic Stages of Datapath

1. IF: inst fetch from inst memory
2. ID: inst decode (decode & read data from reg simultaneously, since inst fields are fixed)
3. EX: ALU
4. MEM: data memory access
5. WB: write back to reg





.

datapath elements

- reg: 32×32 bits reg[0]$\equiv 0$
- pc: hold addr of current inst, the next will be $PC + 4$ or jump
- mem: separate memory IMEM & DMEM, in case of conflict

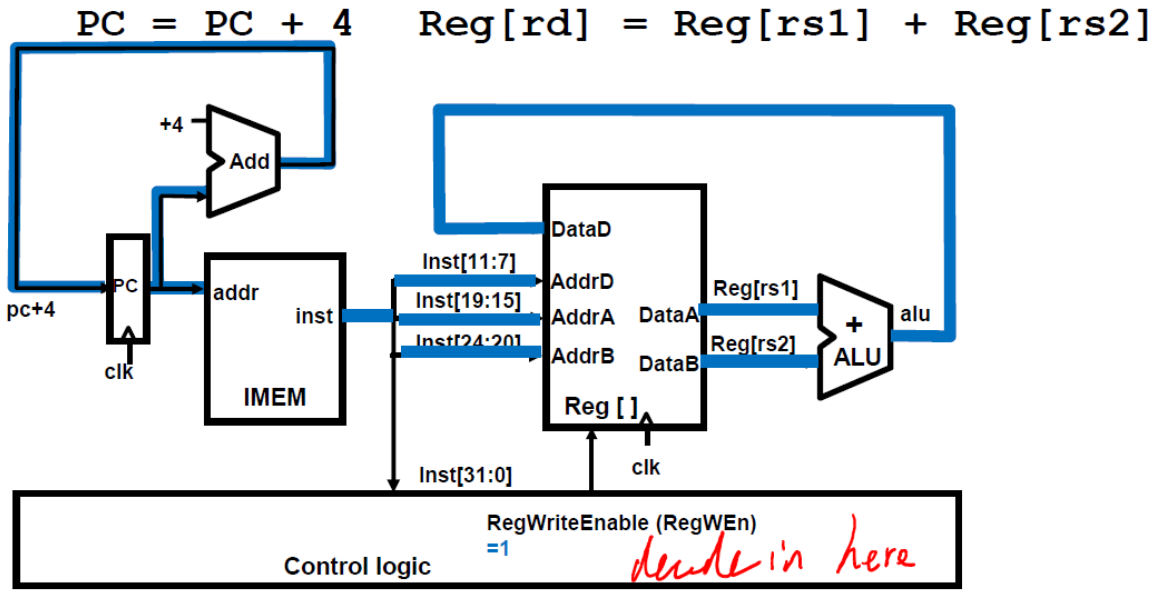
4.1 Datapath Implement

pay attention to critical path timing

no pipeline for now

4.1.1 R-type

pc reg→IMEM→reg read→ALU→reg write

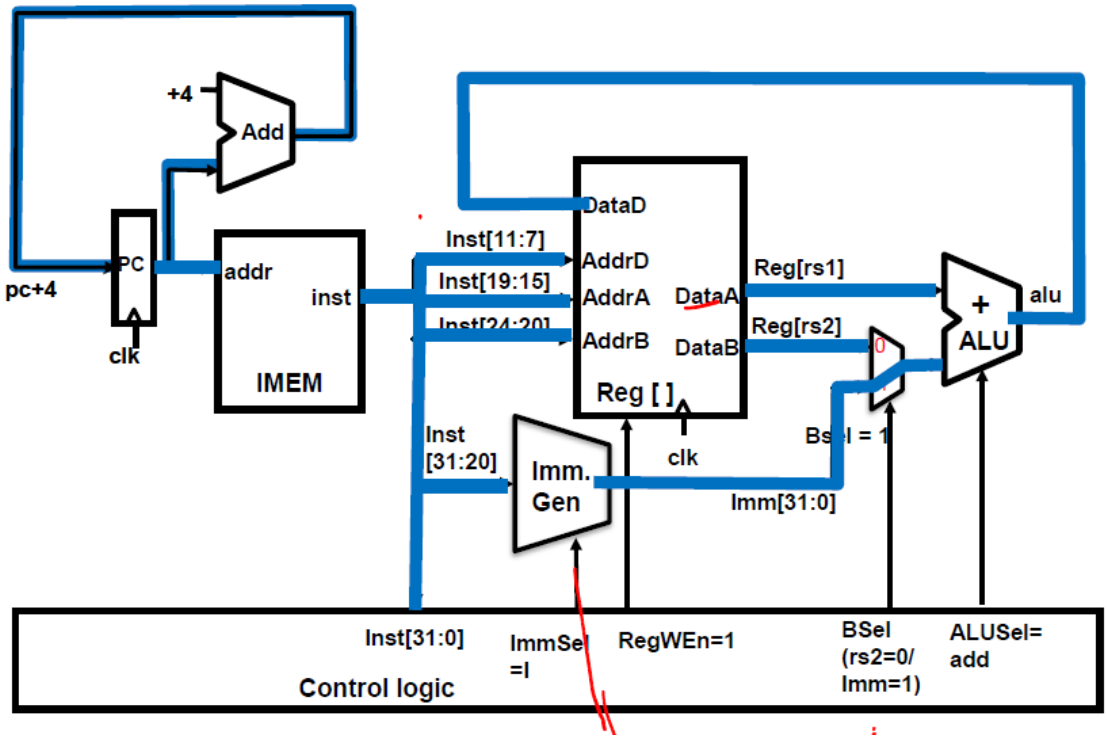


4.1.2 I-type

imm

pc reg→IMEM→reg read & imm gen + mux→ALU→reg write

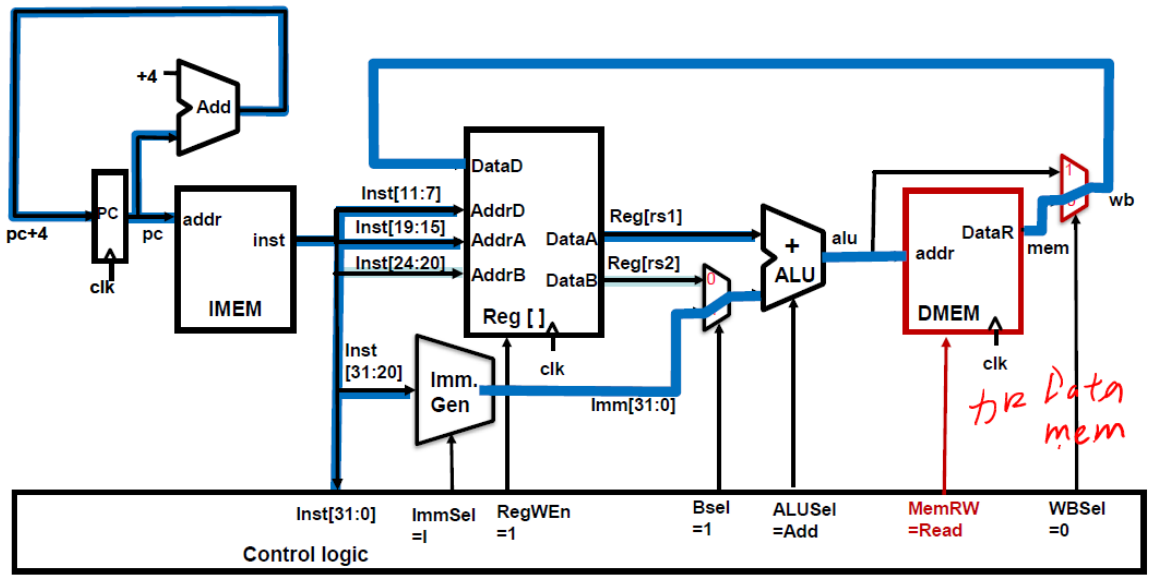
- the second input of ALU (Reg[rs2]) use **imm** instead
- Imm Gen, sign-extend **imm** to 32 bits (copy lower 11 bits & repeat sign bit)



load

need the DMEM

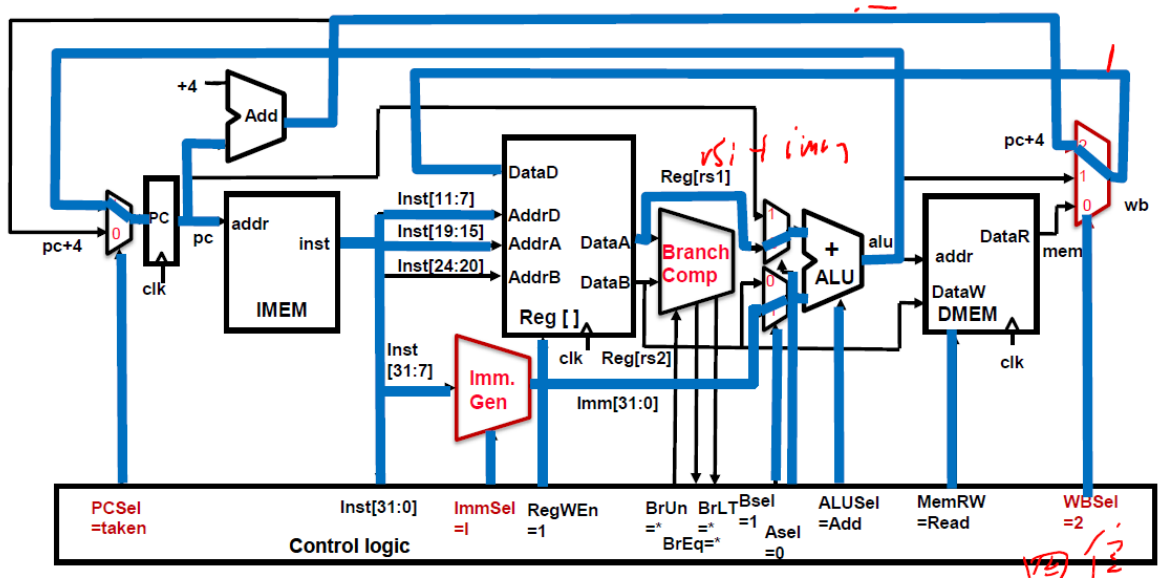
pc reg→IMEM→reg read & imm gen + mux→ALU→DMEM load→mux→reg write



jalr

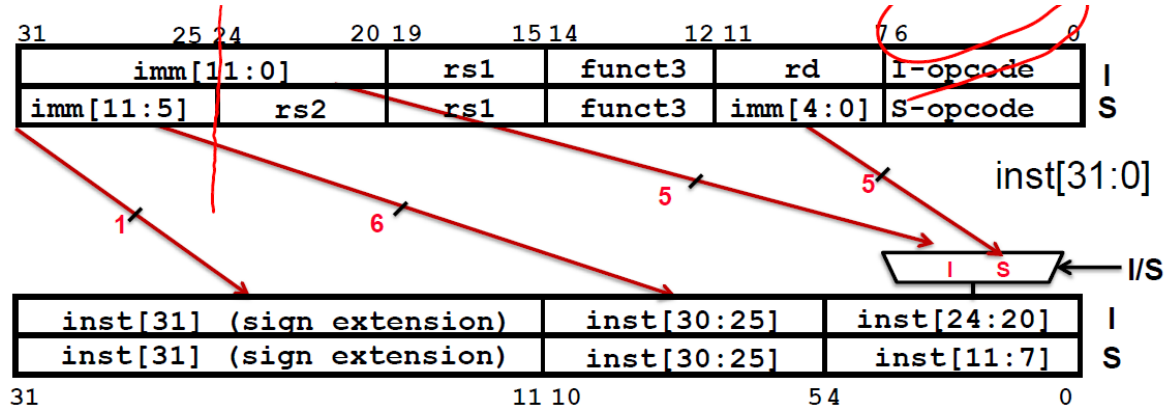
write $PC+4$ to reg, $PC = rs1 + imm$

pc reg→IMEM→reg read + mux & imm gen + mux→ALU→mux

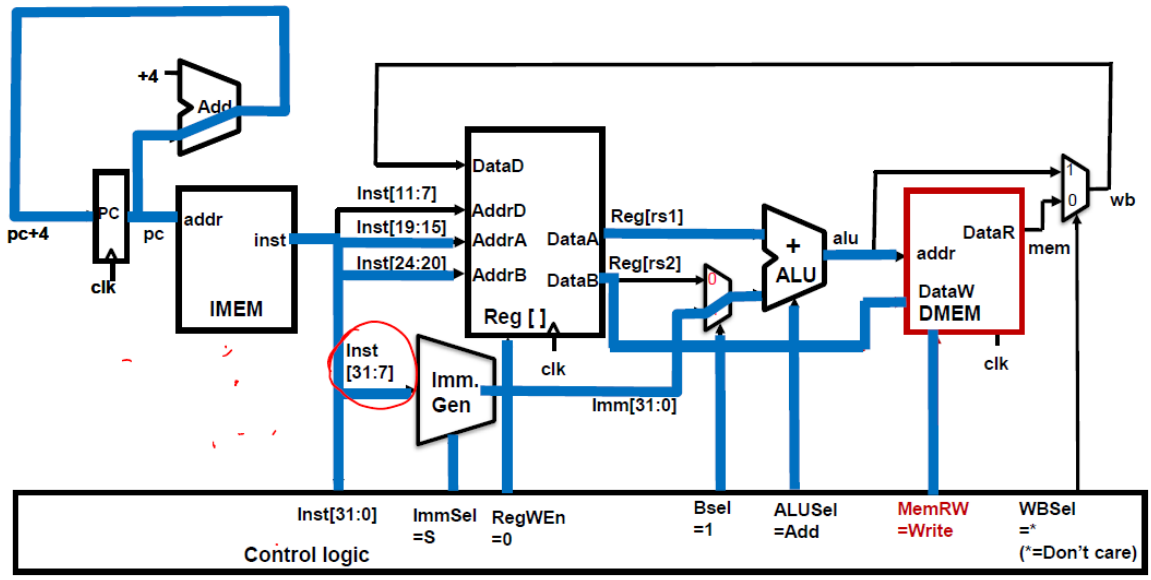


4.1.3 S-type

input more bits of inst into imm gen, only need to choose what is the lower 5 bits of **imm**



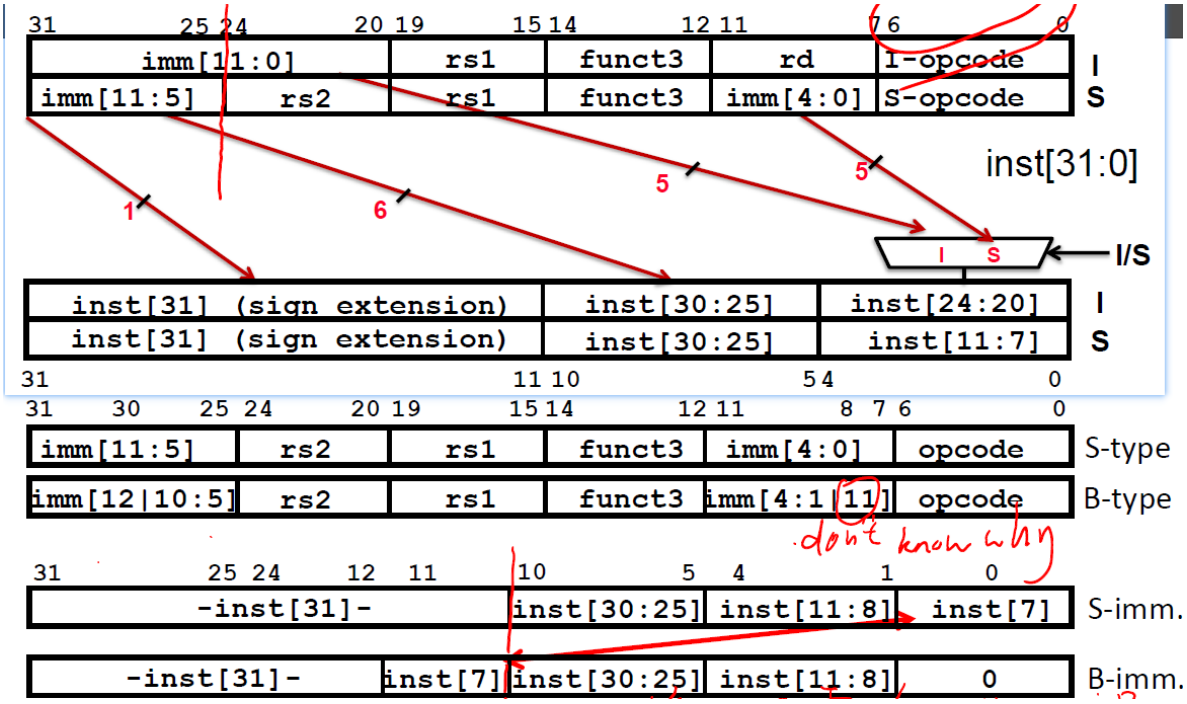
pc reg→IMEM→reg read & imm gen + mux→ALU→DMEM store



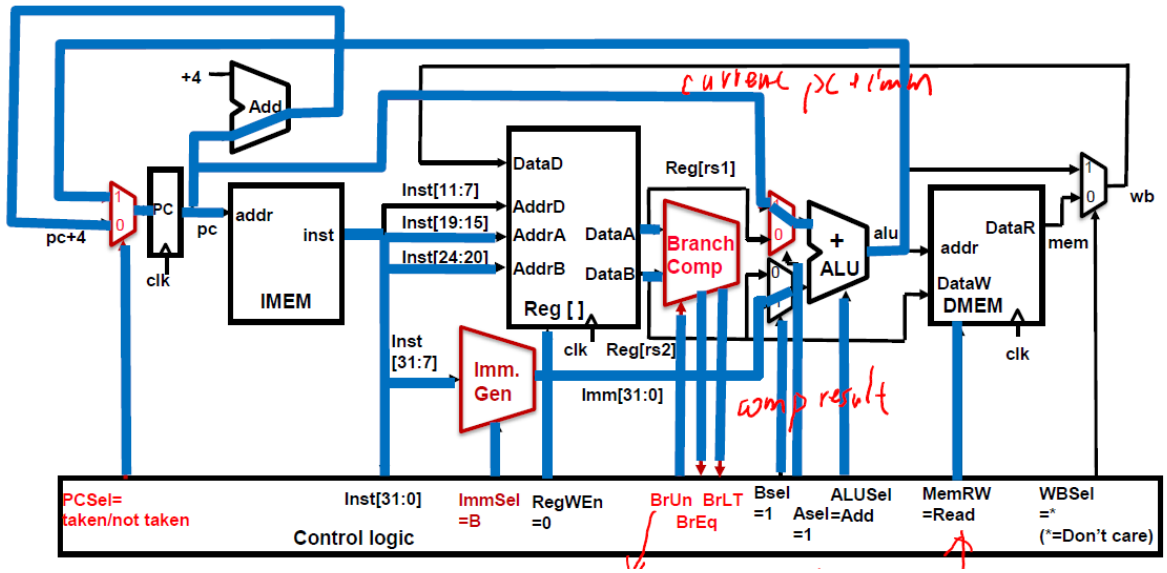
4.1.4 B-type

add a branch comp, decide pc taken or not

more complex imm, but only take lowest bit to replace the 12 bit (imm[11])



pc reg→IMEM→reg read + comp & imm gen + mux→ALU→mux

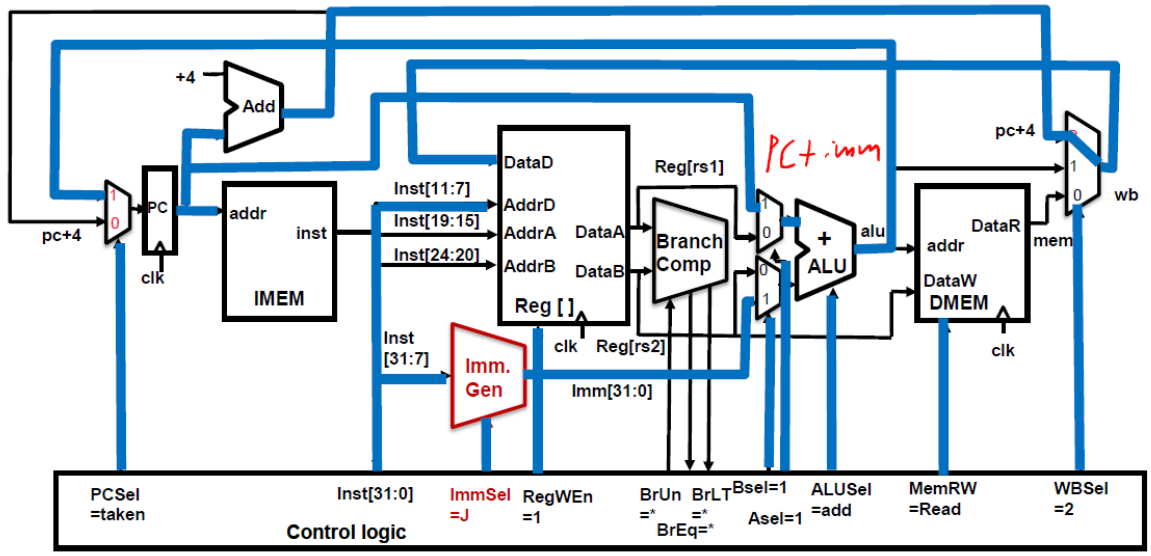


4.1.5 J-type

write $PC + 4$ to reg, $PC = PC + imm$

no reg read ??

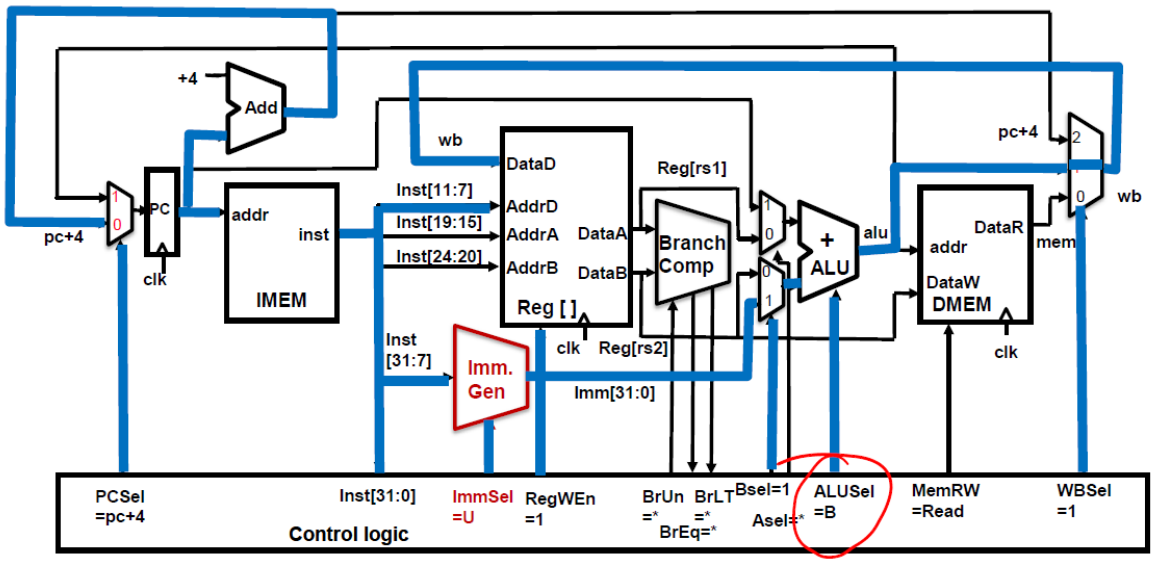
pc reg→IMEM→imm gen + mux→ALU→mux



4.1.6 U-type

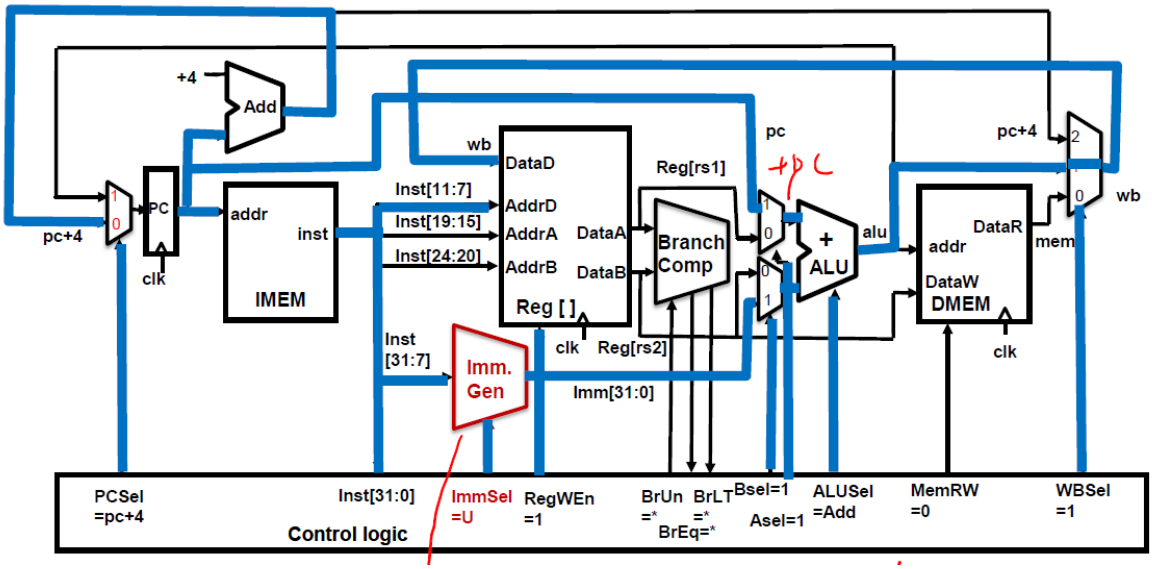
lui

pc reg→IMEM→imm gen + mux→ALU→mux→reg wrtie

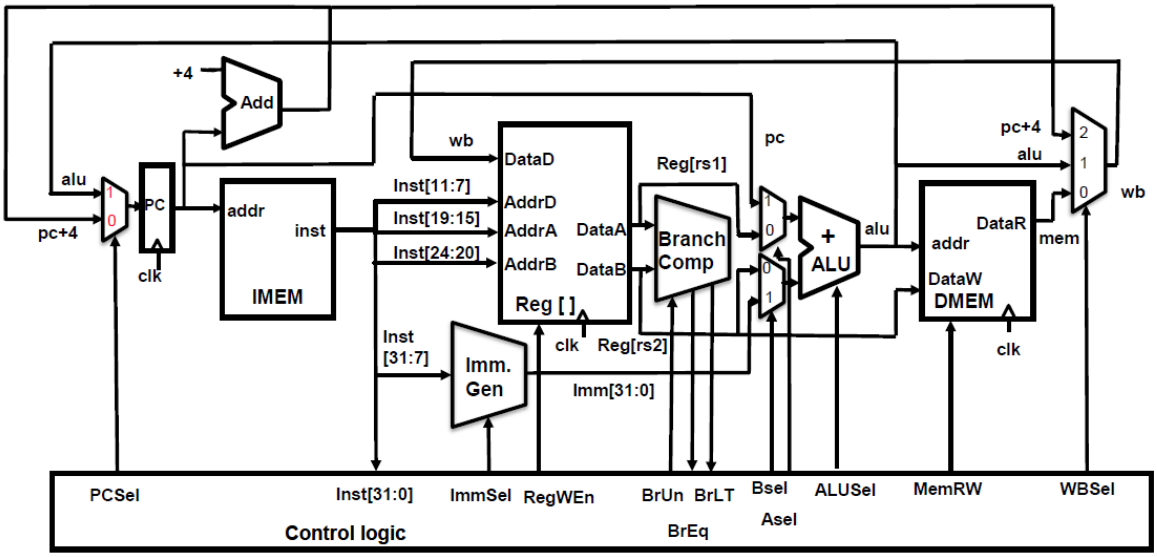


auipc

pc reg→IMEM→imm gen + mux→ALU→mux→reg write



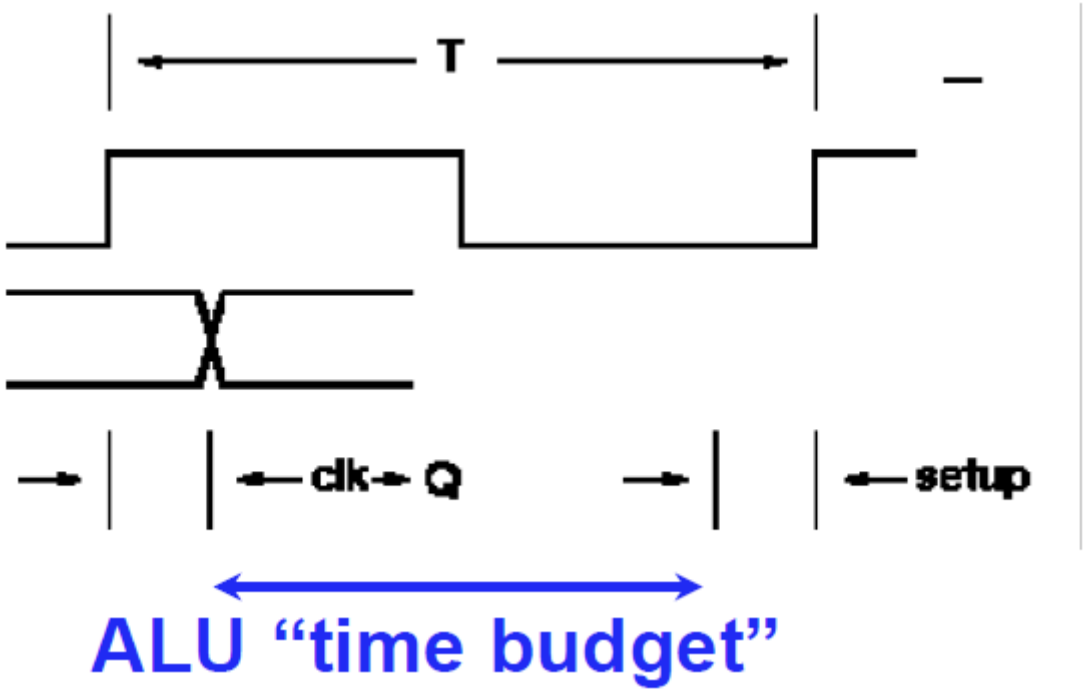
4.1.7 Final Datapath



4.2 Clocking

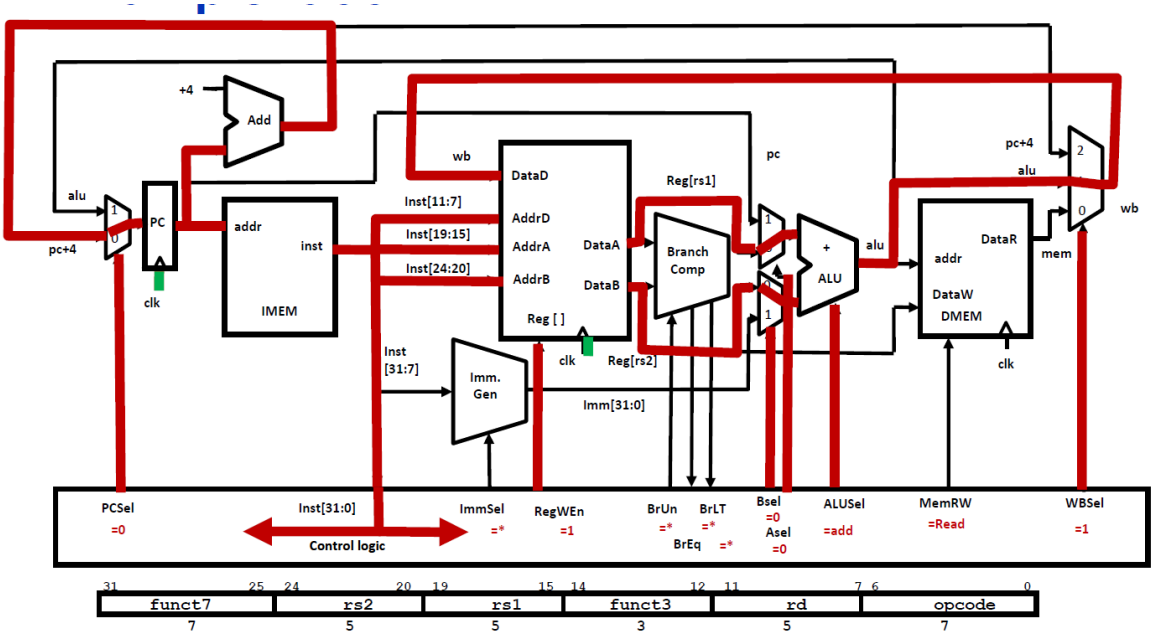
- $T_{clk \to Q}$: time between clk & output Q (read out from reg)
- T_{setup}: input need to change before clk comes (write into reg)
- T_{CL}: combination logic
- T_{skew}: clk will delay

$$T \geq T_{clk \to Q} + T_{+}CL + T_{setup} + T_{skew}$$

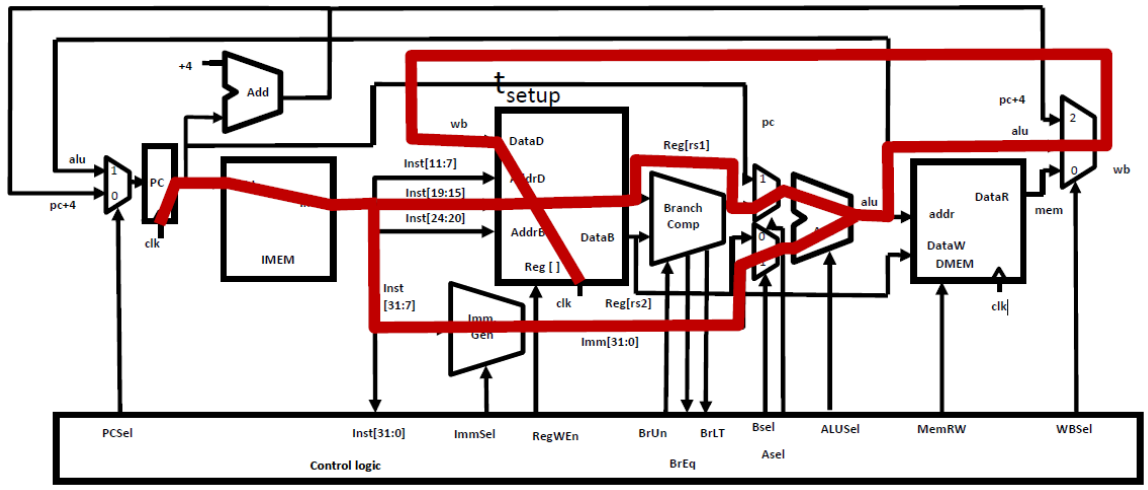


4.2.1 Critical Path

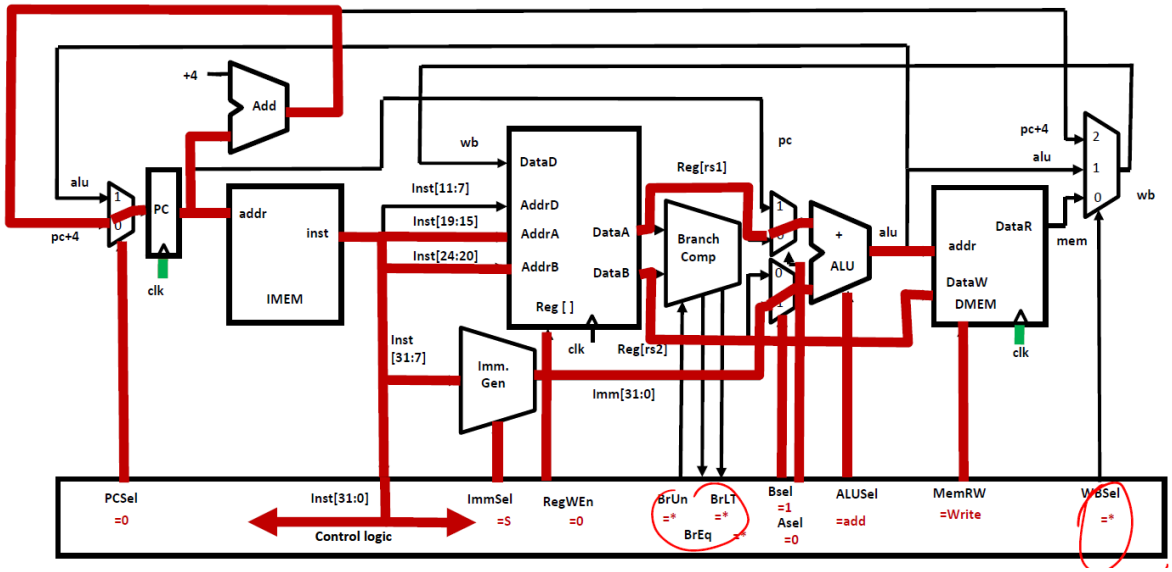
add


+4
Add
alu
pc+4
PC
clk
addr
inst
IMEM
wb
DataD
AddrD
AddrA
AddrB
Reg [ ]
DataA
DataB
clk
Inst[11:7]
Inst[19:15]
Inst[24:20]
Inst [31:7]
Imm. Gen
Imm[31:0]
Reg[rs1]
Reg[rs2]
pc
Branch Comp
+
ALU
alu
addr
DataW
DMEM
DataR
mem
clk
pc+4
alu
wb
PCSel =0
Inst[31:0]
Control logic
ImmSel =*
RegWEn =1
BrUn =*
BrLT =*
BrEq =*
Bsel =0
Asel =0
ALUSel =add
MemRW =Read
WBSel =1
31 25 24 20 19 15 14 12 11 7 6 0
funct7 rs2 rs1 funct3 rd opcode
7 5 5 3 5 7

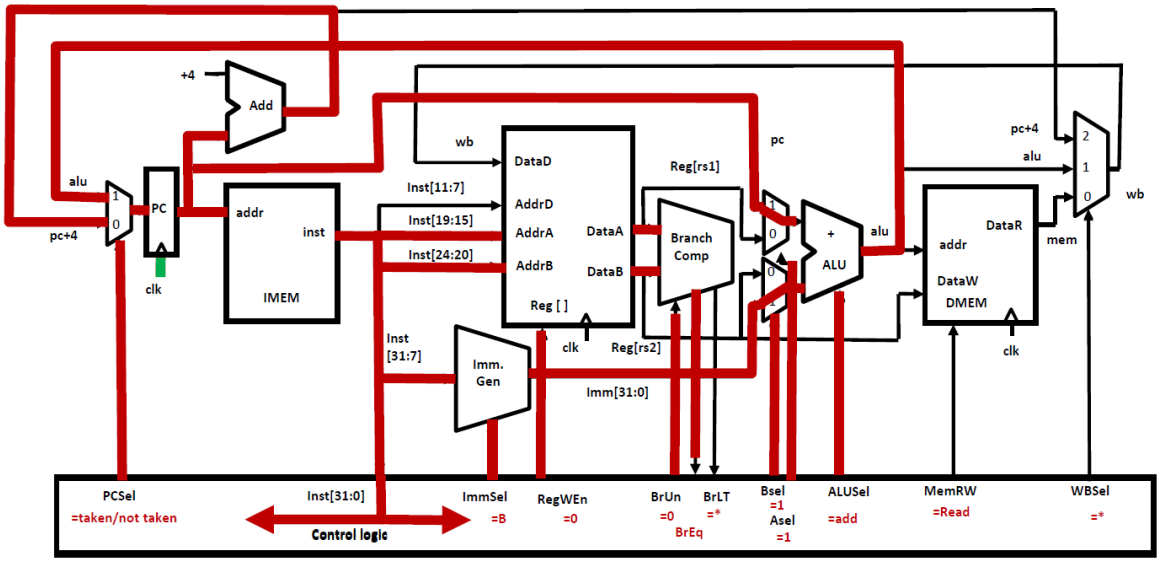
addi


+4
Add
alu
pc+4
PC
clk
IMEM
wb
DataD
AddrD
AddrA
AddrB
Reg [ ]
DataB
clk
Inst[11:7]
Inst[19:15]
Inst[24:20]
Inst [31:7]
Imm Gen
Imm[31:0]
Reg[rs1]
Reg[rs2]
pc
Branch Comp
alu
addr
DataW
DMEM
DataR
mem
clk
pc+4
alu
wb
PCSel
Inst[31:0]
Control logic
ImmSel
RegWEn
BrUn
BrLT
BrEq
Bsel
Asel
ALUSel
MemRW
WBSel

sw


+4
Add
alu
pc+4
PC
clk
addr
inst
IMEM
wb
DataD
AddrD
AddrA
AddrB
Reg [ ]
DataA
DataB
clk
Inst[11:7]
Inst[19:15]
Inst[24:20]
Inst [31:7]
Imm. Gen
Imm[31:0]
Reg[rs1]
Reg[rs2]
pc
Branch Comp
+
ALU
alu
addr
DataW
DMEM
DataR
mem
clk
pc+4
alu
wb
PCSel =0
Inst[31:0]
Control logic
ImmSel =S
RegWEn =0
BrUn =*
BrLT =*
BrEq =*
Bsel =1
Asel =0
ALUSel =add
MemRW =Write
WBSel =*

beq



lw takes the longest, including IF, ID, ALU, MEM, WB

Instr	IF = 200ps	ID = 100ps	ALU = 200ps	MEM=200ps	WB = 100ps	Total
add	X	X	X		X	600ps
beq	X	X	X			500ps
jal	X	X	X			500ps
lw	X	X	X	X	X	800ps
sw	X	X	X	X		700ps

5 Pipeline

5.1 Performance

Iron Law

$$\frac{\text{time}}{\text{program}} = \frac{\text{inst}}{\text{program}} \times CPI \times \frac{\text{time}}{\text{cycle}}$$

- inst per program
  - task
  - alg
  - language
  - compiler
  - ISA
- CPI
  - ISA
  - CPU architecture (superscalar)
- time per cycle (CPU freq)
  - tech
  - power
  - CPU architecture

5.2 Pipeline

term

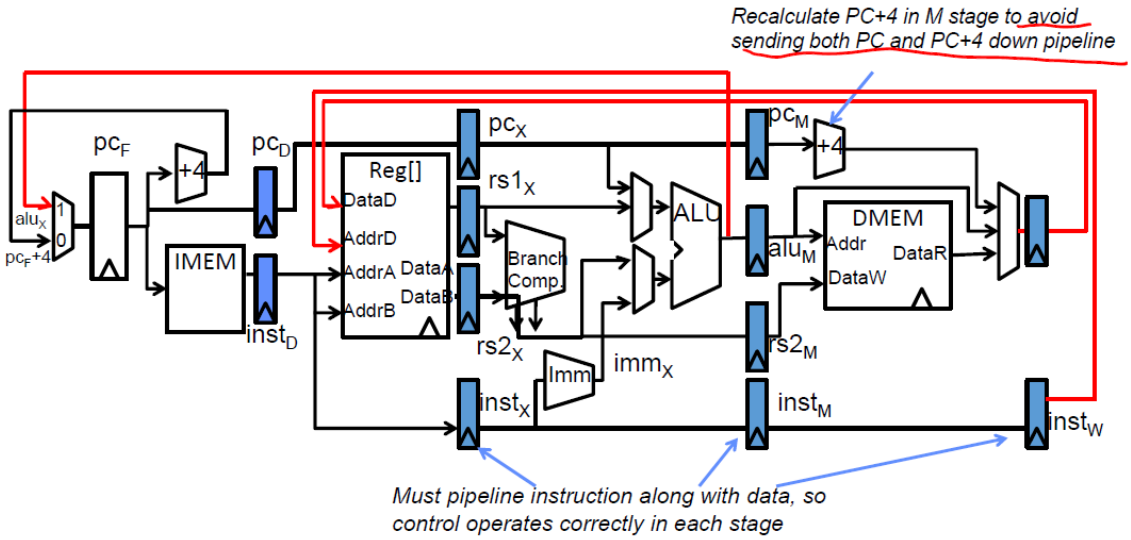
- latency: # of pipeline stage

- throughput: work done in unit time
- execution time (response time): time between start & end of a task

intro

- does not help latency (execution time) of a single inst
- inc total throughput
- potential speedup ≈ # pipeline stage
- inc clock rate (determined by the slowest stage)
- throughput is determined by clock cycle (clock rate)
- # of pipeline stage affect latency not throughput

add reg to datapath to store value



5.3 Hazard

5.3.1 Structrual Hazard

multiple inst compete for one physical resource

- take in turns, add stall
- add more hardware

reg & mem: add more ports, read & write at the same time

5.3.2 Data Hazard

data dependency

- reg: write & read to the same addr

  want Read After Write

- ALU result needed for following inst
  - insert stall (NOP)
  - forwarding, need extra connection in datapath

lw can not forwarding, must insert a NOP, **load delay slot**

can re-schedule codes

5.3.3 Control Hazard

branch not taken, need to flush data from pipeline and convert to NOP

branch prediction

6 Cache

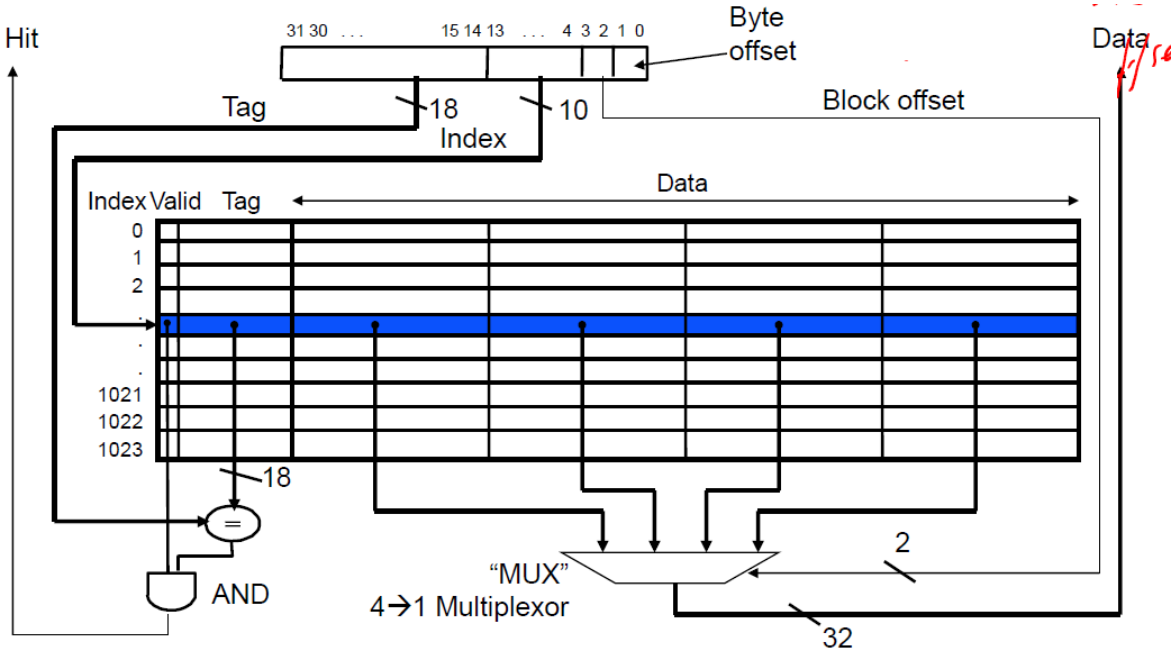
- temporal locality: keep most access data close to CPU
- spatial locality: take data nearby at the same time

6.1 Direct Map

6.1.1 Addr

each cache line contains

- block: store data
- valid bit: indicate whether is empty
- tag: for specify data



addr

- index: specify which row of cache line
- offset: specify which byte in the block
- tag: distinguish addr map into same index

$$\# \text{ block} = \frac{\text{cache size}}{\text{block size}}$$

$$\# \text{ bit of offset} = \log_2(\text{block size})$$

$$\# \text{ bit of index} = \log_2(\# \text{ block})$$

$$\# \text{ bit of tag} = 32 - \# \text{ bit of index} - \# \text{ bit of offset}$$

6.1.2 Accessing Cache

term

- cache hit: block is valid and contains addr (not empty & same tag)

- cache miss: empty or different tag
- block replacement: cache miss with wrong tag, fetch data from mem & replace old one

procedure

1. get tag, index, ofs from addr
2. use index find the block
3. valid or not
4. same tag or not
5. use ofs to find the byte

6.1.3 Write Policy

hit

- write through: update cache & mem
- write back: only update cache
    - add dirty bit to cache line, dirty when is written
    - when a dirty block is replaced, copy data back to mem

miss

- no write allocate: only write to mem, no need to find a block
- write allocate: find a block, fetch data in block, then write the block

6.1.4 Block Size

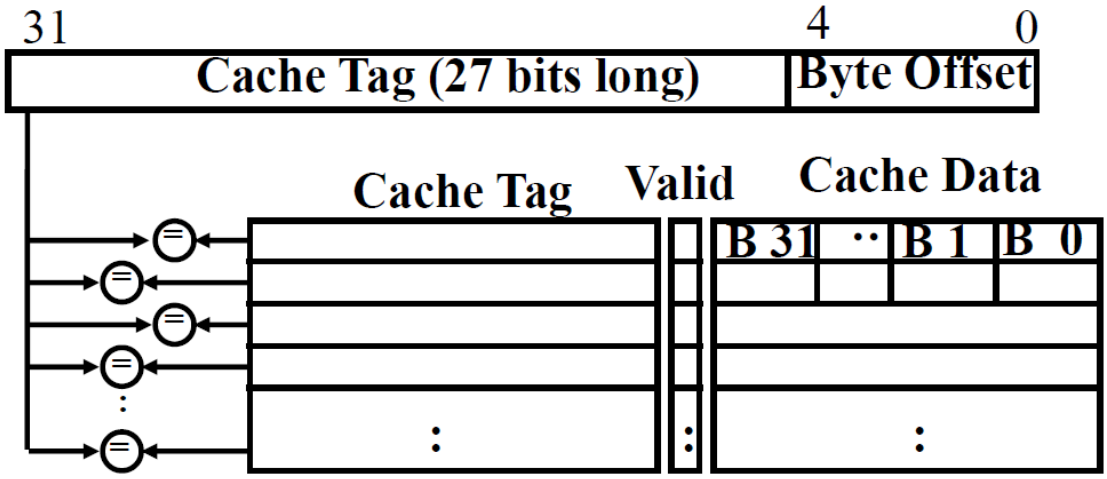
larger block size

- pro
    - spatial locality, more nearby
    - suitable for inst access
    - suitable for sequential array access
- con
    - larger miss penalty, load for a long time
    - when too large few blocks, inc miss rate, more conflict
    - less tag

6.2 Fully Associative

no index, fill in block that is empty, compare tag with all blocks

- pro
    - no conflict with the same index
    - hard to implement



6.3 Type of Miss

3C

- compulsory: access a empty block
  - at least one compulsory miss for each block
- conflict: different addr map to same block (in direct map)
  - bigger cache size (more blocks avoid same index)
  - not occur in fully associative
- capacity: limited cache size (in fully associative)
  - bigger cache size (more blocks to fill in)

6.4 Set Associative

N-way set associative

index specify a set (a set with N blocks), compare tag with all blocks in a set

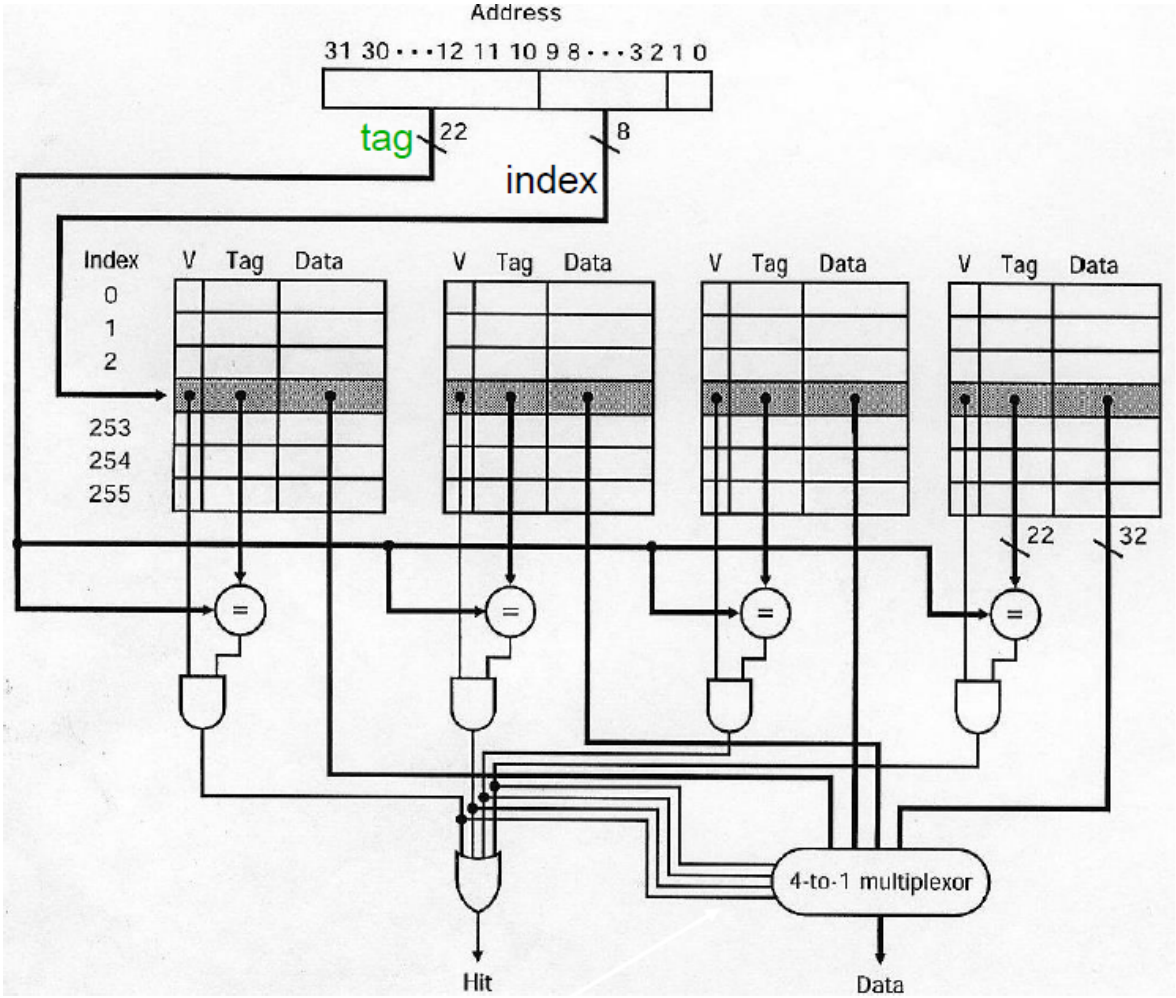
$\#\text{ set} = \frac{\text{cache size}}{N \times \text{block size}}$

procedure

1. get tag, index, ofs from addr
2. use index to find set
3. compare tag with all tag in the set
4. hit?
5. use ofs the find the byte

direct map with respect to set, fully associative with respect to N blocks in a set

- 1-way set associative: direct-map, # block set
- # block-way set associative: fully associative, 1 set



6.4.1 Replacement Policy

for incoming block

- valid bit off? write to first invalid
- all are valid, need to pick one to replace

LRU

- replace the block least recently accessed
- pro
  - temporal locality
- con
  - # bit of LRU is too much

FIFO & Random: easy

6.5 Performance

average memory access time **AMAT**

$AMAT = \text{hit time} + \text{miss rate} \times \text{miss penalty}$

multi-level cache

lower level cache miss penalty is AMAT of higher level cache

6.6

Note
how to deal with 3KB cache?

- 4KB 4-way set associative, block size 128B
- 3 bits index, 7 bits offset
- discard 4^{th} way, always reports miss & never receive data

6.6.1 Associativity

inc associativity

- pro
  - eliminate conflicts
  - suitable for multi-processor
- con
  - each cache line need more bits for tag
  - comparators hard to implement

performance

- larger cache size
  - dec capacity & conflict miss
  - inc hit time
- higher associativity
  - dec conflict miss
  - inc hit time
- larger block size
  - dec compulsory & capacity miss
  - inc conflict miss & miss penalty

7 VM

term

- page: block in VM
- page fault: VM miss

Page Table: convert VA to PA

physical addr space is in DRAM, virtual addr space is imaginary (mapped to physical addr space)

7.1 Translation

- VA: VPN + offset
- Page Table Entry (PTE): extra bits + PPN
- PA: PPN + offset

extra bits:

- valid bit: page in main mem or not
- dirty bti: any word in page is written
- ref bit: set when a page is accessed, clear regularly by OS (for LRU)

7.2 TLB

addr translation is too slow, access PTE + access data

use TLB (cache) of Page Table in cache

use VA to access TLB & Page Table at the same time

- TLB entry = PTE
- fully associative