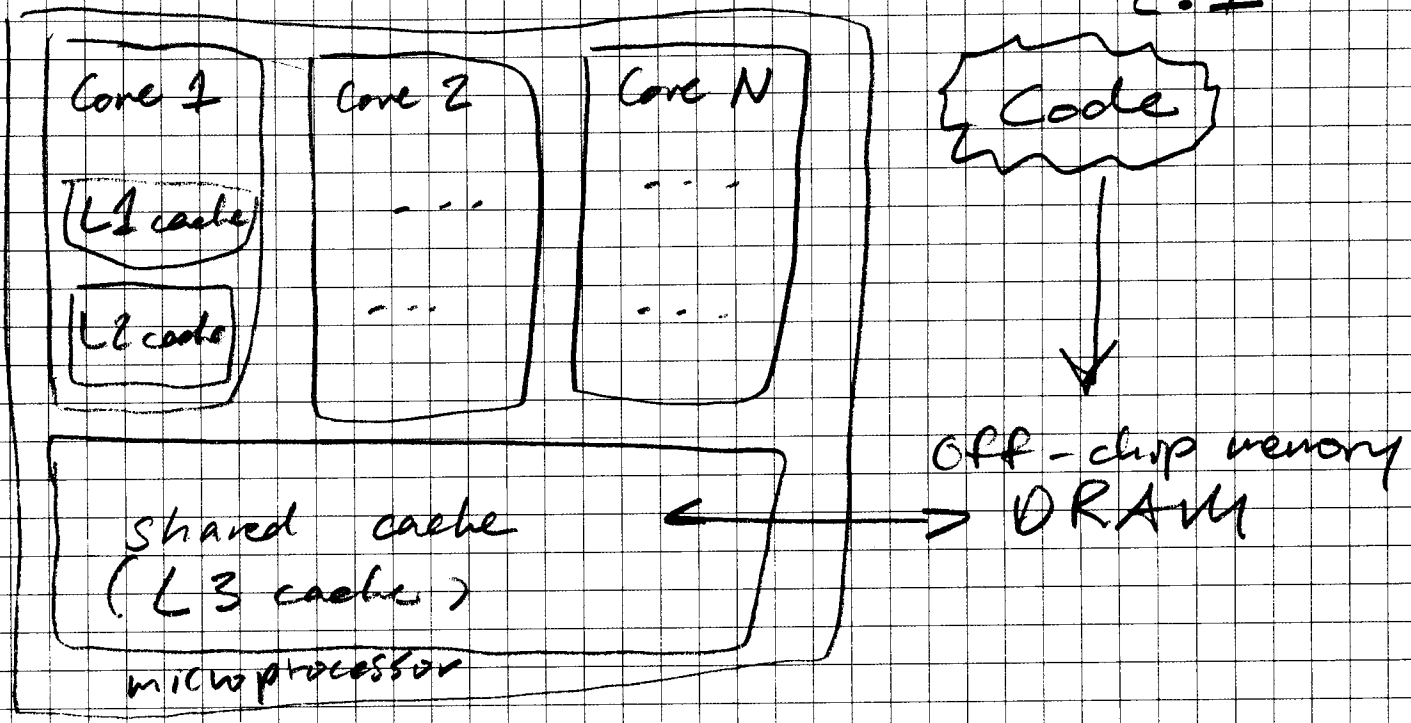
Computer Architecture

8.26.25
L.1



Alba Core Instruction Classes

- Memory
- Arithmetic
- Conditionals

RISC-V: new more widely used instruction set

Ex.

```
if (x == 10):
    y = x + z;
else:
    y = x - z;
```

```
    (x) (10)
    sub r1, r5, r6
    bz r1, IF_BLOCK
(ELSE_BLOCK):
    sub r7, r5, r8
    br CONT
IF_BLOCK:
    add r7, r5, r8
CONT: quit
```

→

```
if (x == 10) goto IF_BLOCK;
ELSE_BLOCK:
    y = x - z;
    goto CONT;
IF_BLOCK:
    y = x + z;
CONT:
```

→ RISC-V in slides

ISA Review, Performance Intro

8.28.25
L02

alsaCore Review:

```
for (i=0; i<10; i++)
  A[i] = A[i] * 2;
```

(goto)

```
i = 0;
Loop: if (i<10) goto Loop_Body;
      goto DONE;
Loop_Body: A[i] = A[i] * 2;
           i++;
           goto Loop;
DONE:
```

(alsaCore)

```
ldi r5, 0      // i
ldi r8, 1      // constant 1
ldi r6, 10     // constant 10

ldi r7, high(A)
ldi r11, 8
shl r7, r7, r11
ldi r11, low(A)
or r7, r7, r11    // &A

Loop:
sub r1, r5, r6
bn r1, Loop-Body
br DONE
Loop-Body:
add r13, r7, r5   // &A = 0x12
ld r2, r13, 0     // loads the array element
shl r2, r2, r8    // 0011 << 1 → 0110
st r2, r13, 0
add r5, r5, r8
br Loop
DONE: quit
```

Five Stages of Instruction Processing:

	add	lol
1. (Instruction) Fetch	✓	✓
2. Decode (Register Fetch)	✓	✓
3. Execute	✓	✓
4. Memory	—	✓
5. Write Back	✓	✓
	4 CC	5 CC

IF - read instruction from mem @ PC

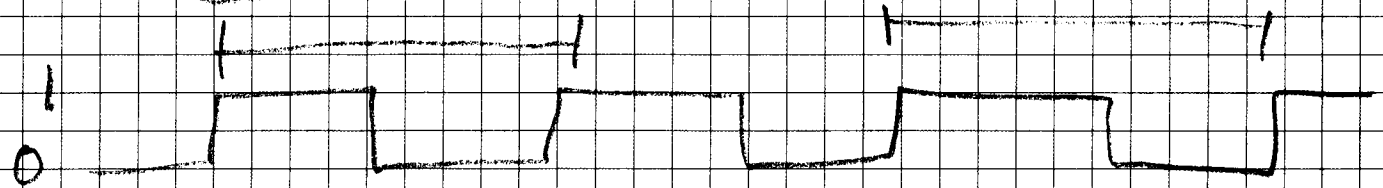
ID - read registers rs, rt, use opcode to determine next steps

EX - arithmetic, logic, comparison operations

MA - access memory for load, store

WB - write results to register file

Don't forget about clock!



Each instruction takes 1 CC

When is the PC updated?

during the last stage of instruction processing

Can we reference memory in 1 CC?

no, since processor and memory reference has a delay

Performance

9.2.25
L05

Recall. The alsacore Datapath and Controller!

= these both exist in RISC-V too

What we know so far: from CD FOEUB

1. How to write assembly code
2. How instructions are processed (generally)
3. " " (in alsacore)
4. How datapath + controller scale 2-5

Soon. These 4 in RISC-V (mostly 1-2)

Now. Performance!

What is performance in context of a processor?

- Time to execute a given instruction
- # instructions it can execute in a given time
- clockrate (time to execute a given program)
- memory usage (e.g. O)
- power

CPU Time: Execution time of a program of interest (top metric for performance)

$$\text{CPU Time(s)} = \text{Instruction Count} \times \frac{\text{cycles per instruction}}{} \times \frac{\text{cycle}}{\text{time}}$$

↓ verbose units

$$\frac{\text{seconds}}{\text{program}} = \frac{\#\text{instructions}}{\text{program}} \times \frac{\text{average cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{clock cycle}}$$

note. units cancel! think chemistry units

note. cycle time inverse of clock rate

$(1 \times 10^{-9} \frac{\text{sec}}{\text{cycle}})$ 1ns ⟷ 1GHz $(1 \times 10^{9} \frac{\text{cycles}}{\text{sec}})$

Formula.

$$\text{CPU Time} = IC \times CPI \times \text{cycle time}$$

Ex.

Instruction class	CPI	# instructions
Branch	3	150,000,000
Store	4	185,000,000
Load	5	260,000,000
ALU/R-type	4	225,000,000

spell check a large file, 820,000,000 instructions are needed. To 5 GHz breaks instructions into classes. Clock cycle in the computer is 2.16 GHz

$$2.16\ GHz = 4.63 \times 10^{-10}\ s$$

$$\text{CPU Time} = \frac{820E6}{\text{inst}} \times \left[3CPI \times \frac{150E6\ \text{inst}}{820E6\ \text{inst}} + 4CPI \times \frac{185E6\ \text{inst}}{820E6\ \text{inst}} + 5CPI \times \frac{260E6\ \text{inst}}{820E6\ \text{inst}} + 4CPI \times \frac{225E6\ \text{inst}}{820E6\ \text{inst}} \right]$$

branches contributions

$$\times\ 4.63E{-}10\ s$$

$$\text{CPU Time} = 1.57\ s$$

Ex. Look for shortcuts

What is CPU Time on this alsecane loop?

Assume 10 loop iterations; cycle time = 1 ns

```
loop: subs r10, r10, r12      4
      be   r10, done          3
      add  r0, r1, r2         4
      add  r3, r4, r5         4
      ld   r6, r7, 0          5
      br   loop               3
done:
```

2 cycles

23 cycles

10 times

(from FDE manual)

$$\left\{ \frac{23 \text{ cycles}}{\text{iteration}} \times 10 \text{ iterations} + 7 \text{ cycles} \right\} \times 1 \text{ ns}$$

$$= \text{total time} = 237 \text{ ns}$$

Key Point: performance problems often come from not solving a missing piece of the CPU time equation

Speedup:

- CPU time is counter-intuitive...
  LOWER CPU time is better
- But we want higher performance

$$\text{Speedup} = \frac{\text{Time}_{old}}{\text{Time}_{new}}$$

Ex. 820E6 instruction spell check. 25% of all load inst. have an ALU inst. right after. To speed this up, we add a new instruction that takes advantage of this

Instruction Class	Original IC	New IC
3 Branch	150E6	150E6
4 Stores	185E6	185E6
5 Loads	260E6	260E6 * 0.75 = 195E6 ✓
7 New Inst.		260E6 - 195E6 = 65E6 ✓
4 ALU	225E6	225E6 - 65E6 = 160E6 ✓
	Total:	755E6

- The new instruction will replace the 2-instruction sequence (load + ALU)
- It will take 7 clock cycles

lot ... sts ... → ld.ALU ...

$$CPUTime_{new} = \left[\begin{array}{l} 3\,CPI \times 150E6 \\ + 4\,CPI \times 185E6 \\ + 5\,CPI \times 195E6 \\ + 7\,CPI \times 65E6 \\ + 4\,CPI \times 160E6 \end{array}\right] \times 4.63E{-}10\ s$$

$$= 1.509s \checkmark$$

$\frac{cycles}{insts} \cdot insts \cdot \frac{s}{cycle}$

$$Speedup = \frac{time_{old}}{time_{new}} = \frac{1.577s}{1.509s} = 1.04$$

It good, eh Sad

RISC-V Instructions and Programming

9.4.25
L04

Sidenote: CPI can be interpreted in 2 ways:

1. CPI for a given instruction
   → must be a whole #
2. weighted average CPI used in formula
   → not necessarily a whole #

both just called CPI!

$$\frac{\text{CPU}}{\text{Time}} = IC \times CPI \times \text{cycle time}$$

What about multi-core?

Amdahl's Law: optimize the common case, also applicable to multi-core

$$\text{Speedup}_{\text{overall}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

Speedup Enhanced:

Ex. 1.5× speedup for all ALU insts.

Fraction Enhanced:

10% of program is ALU insts.

$$\text{Speedup}_{\text{overall}} = \frac{1}{(1 - .1) + \frac{.1}{1.5}} = 1.034$$

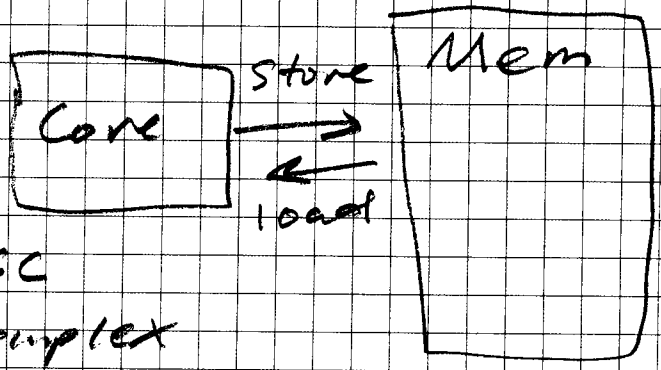
Multicore only as good as its algorithms:

$$\text{Speedup} = \frac{1}{(1 - \text{Fraction}_{\text{parallelized}}) + \frac{\text{Fraction}_{\text{parallelized}}}{N}}$$

N = # of cores

RISC-V Instruction Set Architecture (ISA):

Similarities to alsaCore:



- Load/store model
- RISC approach
  - Reduced Instruction Set Computer
  - CISC → Complex
- Same memory organization (.data)
- Id/st offsets
- assembly code format

Differences (RV32I):

- Data transfers on 32-bit words
- Instructions 32-bit
- 32 registers
- immediates (addi)
- addressing

RISC-V Instructions:

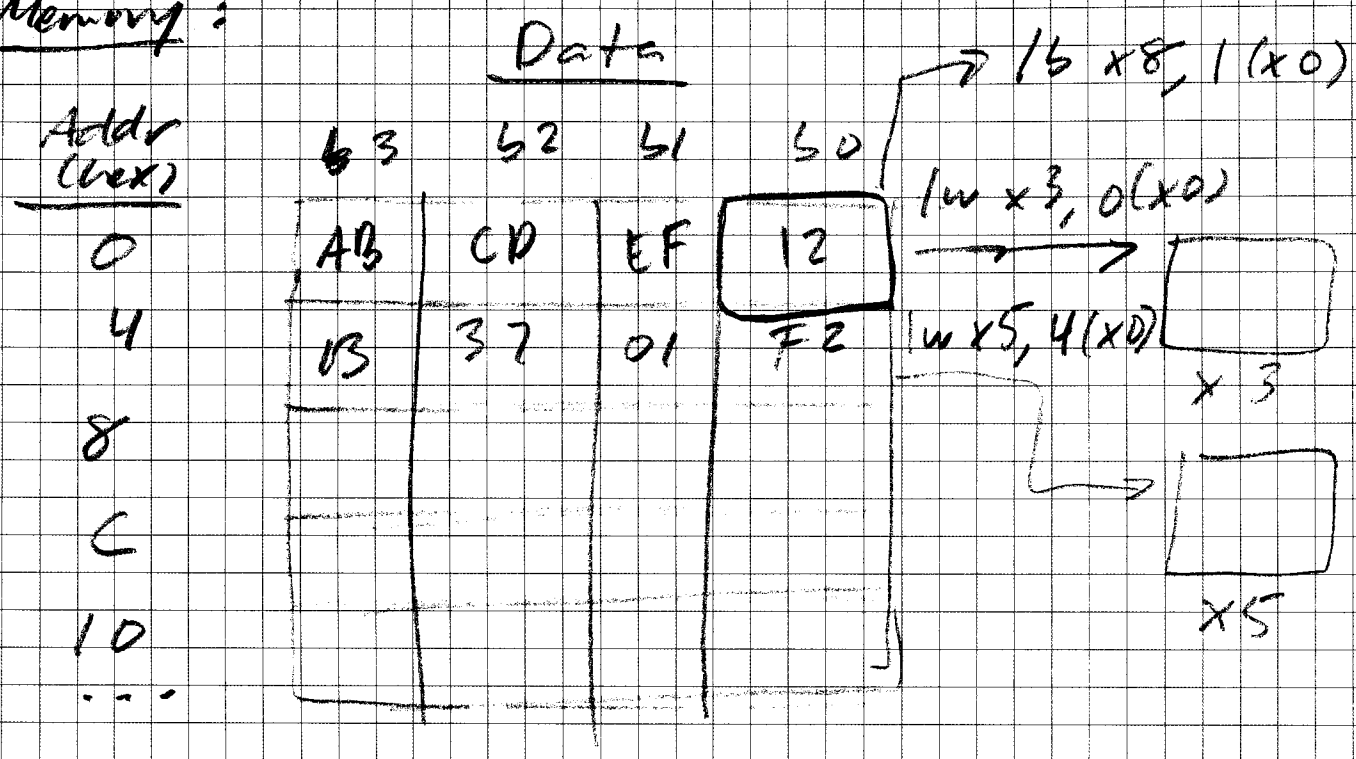
- Listed on the "green" sheet

Registers and Memory:

32 registers: x0 - x31
- x0 always equals 0

2^{30} memory words: Memory[0], Mem[4], ...
- 32-bit words, 4 bytes
  - st → 32b to mem
  - ld ← 32b from mem

Memory:



C to RISC-V:

Ex. // assume i → x10, x → x12, 4 → x14

```
for (i=0; i<5; i++) {
    x = x+4;
}
```

```
li x10, 0
loop: add x12, x12, x14
      addi x10, x10, 1
      blt x10, 5, loop
ebreak
```

Ex. x → x1, a → x10, b → x11, c → Mem[0x40]

```
if ((x>10) && (x<20))
    a = b+c;
```

```
li x15, 11
li x16, 20
blt x1, x15, done   // if x<11
bge x1, x16, done   // if x≥20
lw x17, 0x40(x0)    } if body
add x10, x11, x17   }
done: ebreak
```

RISC-V Instruction Types and Encodings

9.9.25
L05

note.	algalore	RV32I
call a function:	jal insert	jal x1, insert
return:	jr x15	jr x1

ie. you can choose the return register

C to RISC-V Ex:

```
for (i=0; i<10; i++)
    A[i] = A[i] + 2
```

Addr (hex)

A

1000
1004
1008
100C
1010

multiples of 4!

```
li x5, 0        // i
li x6, 10
LOOP:
    slt x5, x6, LOOP_BODY
    j DONE
LOOP_BODY:
    slli x8, x5, 2      // i * 4
    add x8, x7, x8      // &A + i*4
    lw x20, 0x0(x8)     // x20 <- A[i]
    sw x20, x20, -1
    addi x5, x5, 1
    j LOOP
DONE:
```

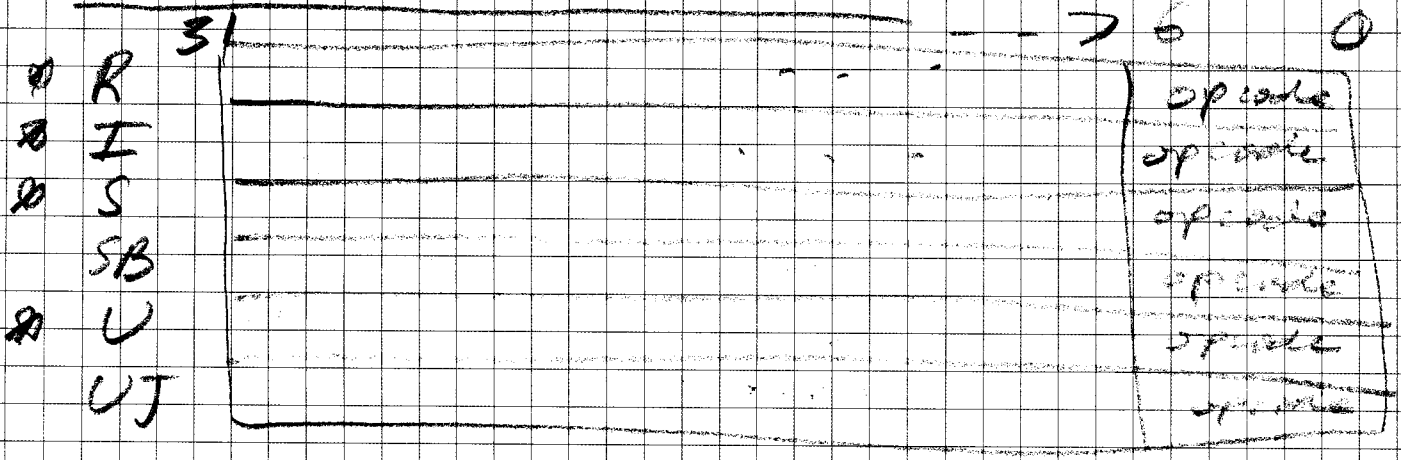
note. register conventions exist!

x0 always 0

RISC-V Instruction Formats:

assembly instructions 1:1 with binary machine code

4 Core Instruction Formats:

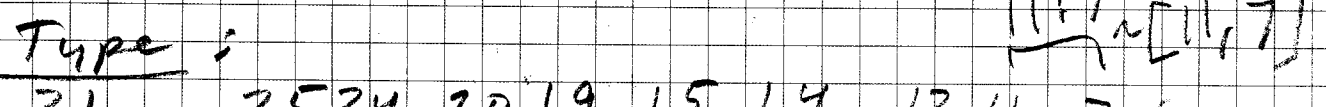


full encoding on green-sheet!

Side. RISC-V is little endian

end of fw/sw not on word boundaries
remainder of word is padded with 0s

R-Type:



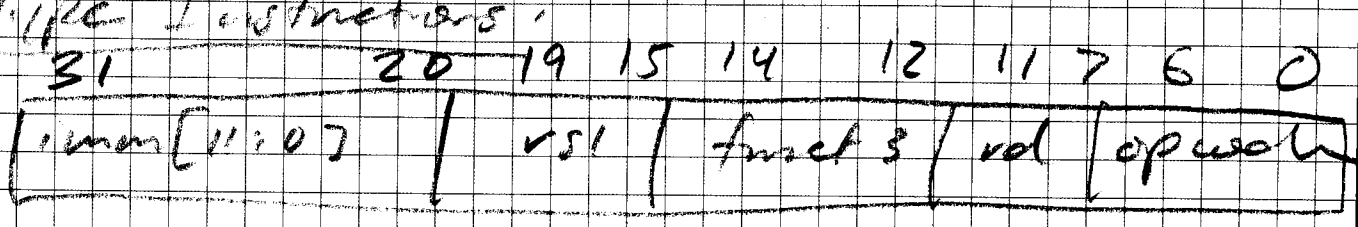
ex. add x7, x10, x11

0000000 01011 01010 000 00111 0110011

0 0 B 5 0 3 B 3

0x00B503B3

I-like Instructions:



ex. lw x5, 0x20(x0)

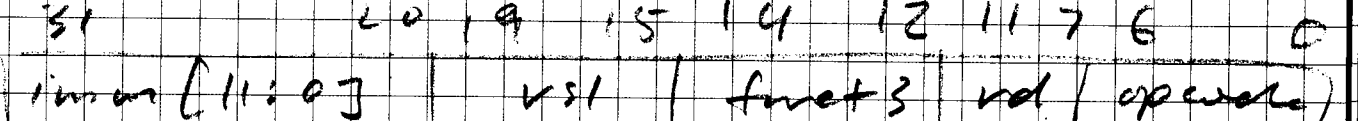
(rd) (imm) (rs1)

0x20 = 00100000 but need to pad

00000010000000000 010 00101 0000011

0x02002283

ex.



addi x7, x10, 0x15

00000001010101010 000 00111 0010011

0x01550393

S-Type Instructions:

31 25 24 20 19 15 14 12 11 7 6 0

S | imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode

ex. sw x5, 0x20(x0)
(rs2) (rs1)

0000 001 | 0 | 0101 | 000 | 00 | 0110 0010 | 0100011

ex. sh x10, 0x100(x2)

0001 | 0000 | 01010 | 00010 | 001 | 0000 | 0100011

0x10508823

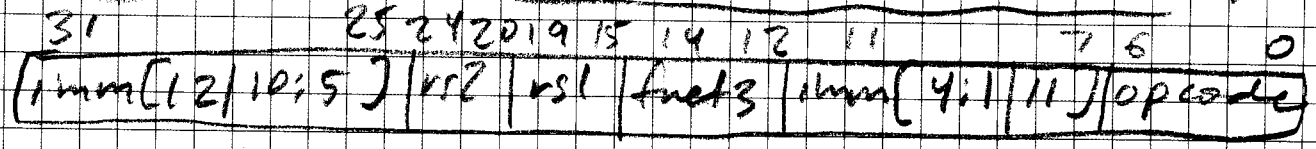
* REMEMBER: immediate bits are split in this type — different pos than in R or I type *

note. store instructions do not modify any registers

RISC-V Control and Procedures

9.11.25
L06

SB-Type Instructions (Conditional Branches):



ex. beq x5, x6, LBL // LBL 4 insts after beq
rs1 rs2

0000000 00110 00101 000 10000 1100011

Note. imm[12:0] = imm13, (13-bit immediate)

0x1000: beq x5, x6, LBL
0x1004
...
LBL: 0x1010

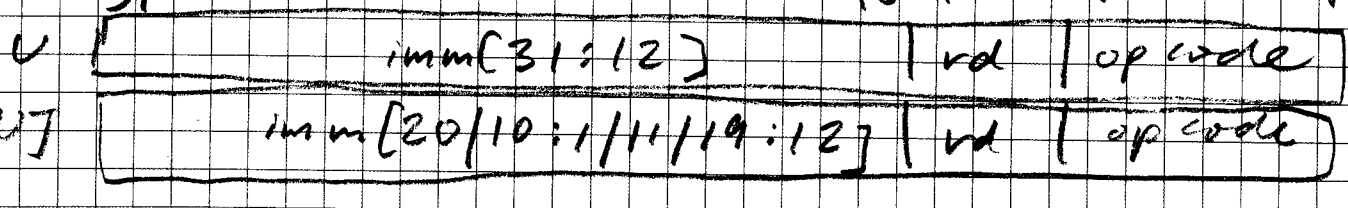
+16

imm[0]

imm[12:10] = 0x 0 00 00000 0 1000 0
12 11 10 5 4 1

Note. imm[0] not encoded since addrs in multiples of 4
imm[1] is encoded since RISC-V supports half-word addresses

U and UJ Instructions:



U-Type insts: auipc, lui

UJ-Type insts: jal

ISA and Datapath Design: these encodings are so messy to **save transistors**

- having as many pieces in the same place results in simpler hardware

Calling Procedures in RISC-V:

jal x1, addFun // addFun is 9 instrs after jal

0 0000001000000000000 00001 1101111
imm[20|10:1|11|19:12] rd opcode

imm[20:0] = 16 = 0 0000 0001 0000
imm0

Calling Conventions: Register Usage

Callee-saved: function being called saves these registers if it wants to use them

Caller Saved: function calling another saves these registers if it wants to use them

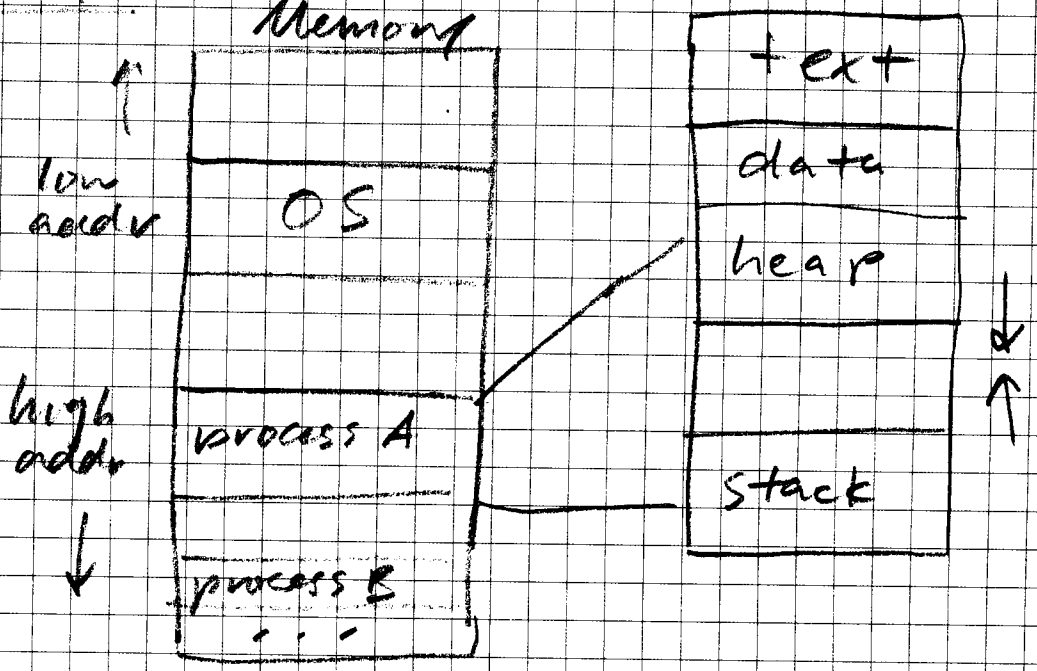
All conventions in green sheet

More Complex Functions:

- There are only 8 "a" registers: x10-x17 → max 8 (a0, a1) return
  a: argument registers
  i.e. void fun(arg0, ..., arg7);
- Need to support (nested/recursive) procedure calls

Solution to both: The Stack!

The Stack:



%rsp = stack pointer

ex. 1: %r11, 0x527 // want to preserve

pushq %r11 ← save (push) %r11 to stack

addl %r11, addl_fn ← we want %r11 to have val 0x527

popq %r11 ← restore (pop) %r11 from stack

addl_fn:

ldi: %r11, 0x500 // %r11 changed!

Pipelining Introduction

9.16.25
[07]

Nested Proc Call Ex:

main:

@ 0x0200: jal x1, funA // x1 = 0x0204

funA: jal x1, funB // x1 = 0x0304

@ 0x300

How many times must the return address reg be saved and restored to the stack?

(1) - on every nested function call

Ex cont:

```
main:
    jal x1, funA
    //...
    ebreak

funA: addi x2, x2, -4 // adjust stack pointer
      sw x1, 0(x2) // push x1 to top of stack
      jal x1, funB
      lw x1, 0(x2) // "pop" x1 from top of stack
      addi x2, x2, 4 // adjust stack pointer
      jr x1

funB  //...
      jr x1
```

Pipelining Analogies

Washer | Dryer | Fold | Put away
20 min | 40 min | 10 min | 5 min

time →

20 40 10 5 20 40 10 5 20 40 10 5

LD1 W D F P
LD2 W D F P
LD3 W D F P

= 225 min for 3 loads

Pipelining: (40) 40 40 40 40 (40) — still 40 to account for other people doing laundry

LD1 W D F P
LD2 W D F P
LD3 W D F P

3 loads = 240 min

* Still only 40 min per cycle so faster at scale!

RISC-V: Datapath and HLSM
- not the focus of this class, but exist

F
D
E
M
W

5-stage pipeline

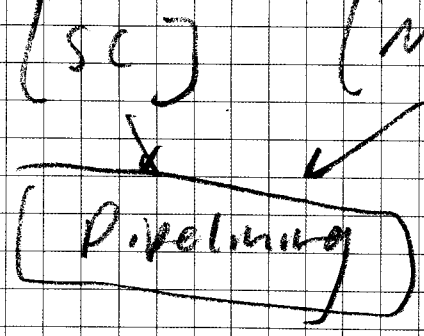
Pipelining Questions and Challenges:

- How deep should the pipeline be? (How many stages?) — Tradeoffs
- How do we evaluate the performance? (is it always better to pipeline)
- Hazards are a significant challenge
  - Structural hazards
    eg. PC+4 and add together?
  - Data hazards
    eg. insts are dependent on each other
  - Control hazards
    eg. branch pathing

Single vs Multi-cycle

Single-cycle: every instruction completes in a single clock-cycle *

multi-cycle: what we've studied so far * clock cycle length of longest stage *



* todo this, the clock-cycle is stored to the length of all stages (FDEMW)

$$CPU\_Time = IC \times \underset{\text{goes down}}{CPI} \times \underset{\text{goes up}}{T}$$

Graphically Representing Pipelining:

- 3 different options — all in slides

Pipelining Performance and Hazards

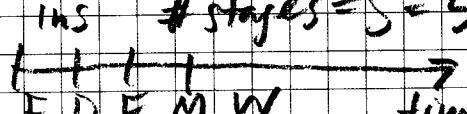
9.18.25
LC8

Pipeline Math (Performance):

Multi-cycle Design:

1) lw FDEMW 2) add FDEW 3) beq FDE

- Assume upperbound/worst case
- Time between instructions fetches = (S)(T)



lw F D E M W
add F D E M W
beq F D E M W

stages ↑ ⇒ time per stage

- Time between instructions fetches = 1 ns = T

$$(5\ \text{stages})\left(\frac{1\ \text{ns}}{\text{stage}}\right) = 5\ \text{ns}$$

$$\text{Speedup} = 5 \quad \left(\frac{T_{old}}{T_{new}} = \frac{5\ \text{ns}}{1\ \text{ns}}\right)$$

But we care about completing instructions...

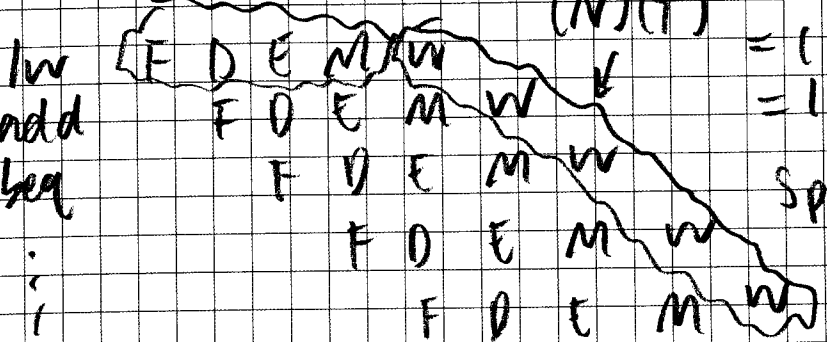
Non-pipelined: (S)(T)(N) = (5)(1ns)(1K) = 5000ns

(# stages) (time per inst (cycle time)) (# insts)

pipelined: (S-1)(T) + (N)(T) i.e. NT + (S-1)T

= (1000)(1ns) + (4)(1ns)

= 1004 ns



$$\text{Speedup} = \frac{5000}{1004} \approx 5$$

Rule. max speedup is # stages

Note. startup time (S-1)(T) → 0 with large N

Dependency: producer-consumer relationship in instruction stream

ex. addi x10, x0, 4 → add dependant on addi
add x10, x10, x10

Hazard: condition in hardware that may lead to stall when a dependency is encountered

add reads after addi writes

called a true data dependency

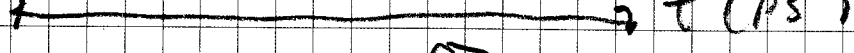
Structural Hazard: add does not access memory, why does it need to go through M stage

lw	F	D	E	M	W	not possible in hardware
add		F	D	E	W	

Solution: 1. 2 register files
2. 2 write ports on register file

ex. R-type → instA, instB → F, E → add 2nd ALU

Data Hazards:



lw x1, 100(x4) — F D E M W

add x10, x1, x10 — F D (stall) E M W (needs this)

beq x10, x12, done — F D (stall) E M W

sources

Solutions: STALL

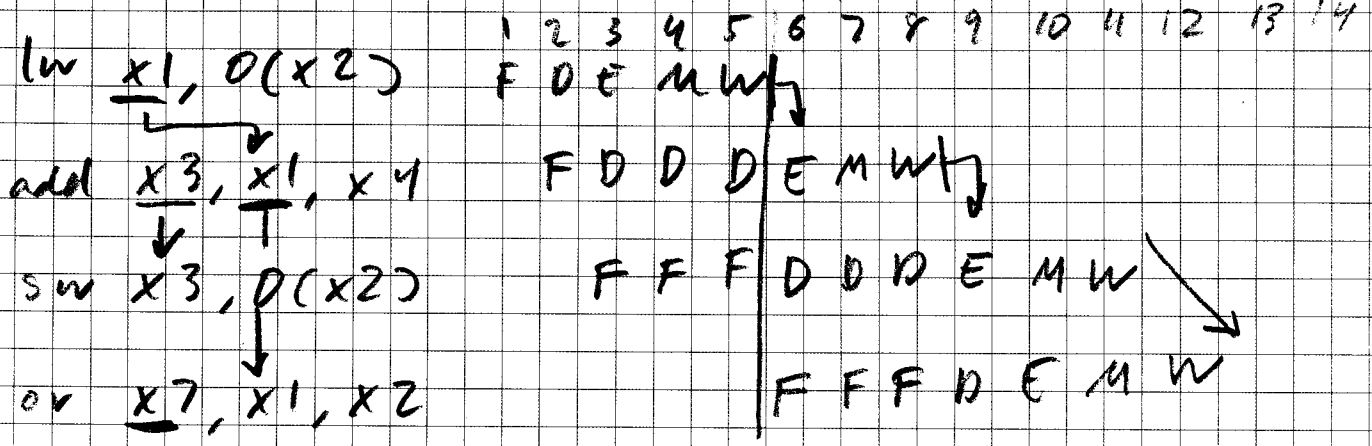
	1	2	3	4	5	6	7	8
lw	F	D	E	M	W			
add			...	D	E	M	W	

Forwarding: pass information as soon as available

Pipe Trace Checklist (No Forwarding): Stalling

1. First instruction always free: F D E M W
2. Work top-down, for each instruction, check for data dependences:
   a) determine producer/consumer relationships
   b) Stall: consumer exits D (enters E) 1 cycle after producer finishes W
3. Sanity check:
   a) No duplicate letters in a single column (for now...)
      - only 1 instruction supported per pipe stage
   b) Duplicate the stage where hazard is detected (no bubbles)
      → bubbles are valid but fraught way to fill pipe trace / depict stalls ... avoid in this class!

Pipe Trace Option 1: Stall



total cycles? 12

Pipe Trace Checklist (Assume Forwarding):

...

2 b) load use:

lw followed immediately by dependent inst

→ 1 cycle stall (no trickle effects)

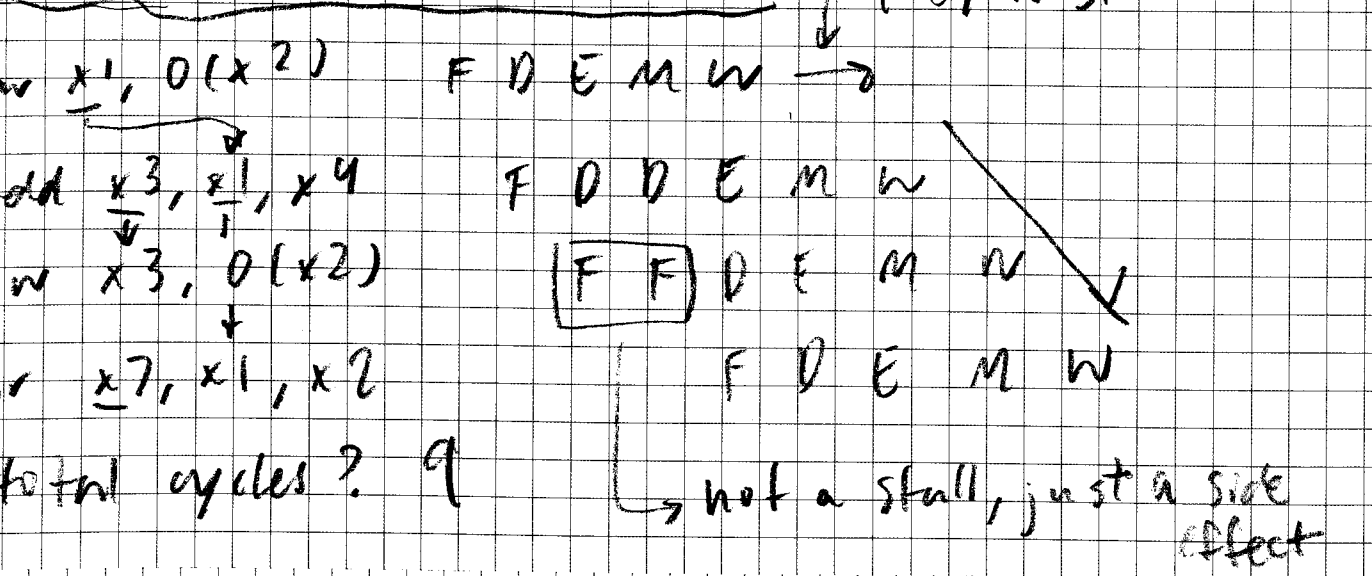
c) any other data dependences can be handled by forwarding (no stall)

i. forwarding: if consumer enters D before producer enters W

ii. sw writes no registers → no dependent insts

...

Pipe Trace Option 2: Forward



total cycles? 9

Pipelining Hazards

9.23.25
Log

note. Endianness

0 1 2 3 4 5 6 7 8

byte 1 byte 0
byte 2
byte 3

RISC-V is little endian!

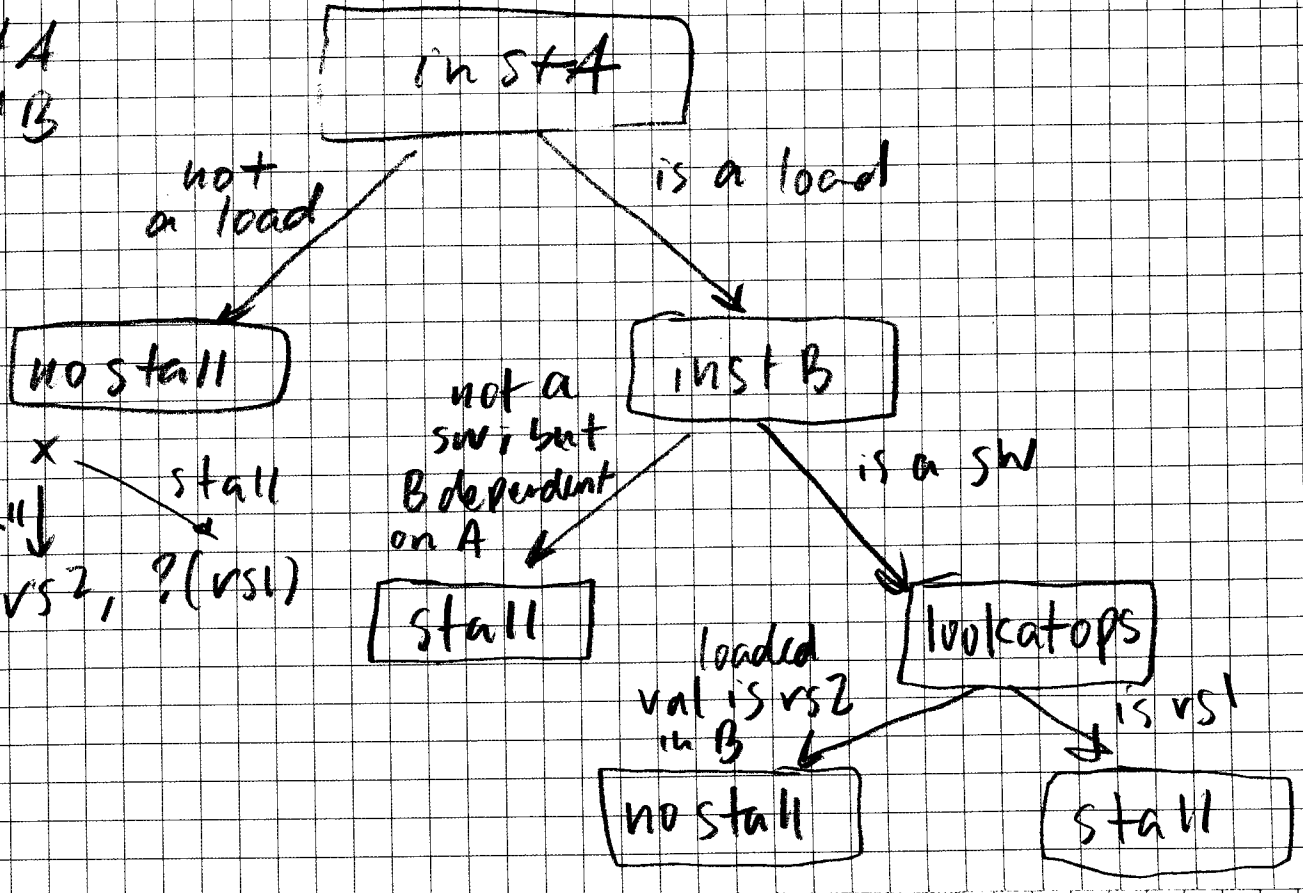
Pipe Tree Option 2: Forward - SW exception

lw x10, 0(x12) F D E M W
sw x10, 0(x11) F D E M W

0x4 ↑ ↖ "x10" stored to memory

When to Stall in Forwarding?

...
inst A
inst B
...



Hazards and Performance:

- Load-use hazard unavoidable in this architecture
  → How does it affect performance?
- Ideal pipelined CPI = 1 (IPC = 1)
  → time to fill pipeline can be ignored

Ex. "Pipe Trees Option 2: Forwarding"

```
F D E M W
  F D D E M W
```

(load-use hazard)

$$\frac{1 \text{ stall cycle}}{4 \text{ insts}} = 0.25$$

Ideal CPI + Stall CPI = 1 + 0.25 = 1.25

Another Ex (with forwarding):

```
A sw x4, 0(x8)        F D E M W
B add x3, x4, x8        F D E M W
C lw x9, 0(x3)            F D E M W
D sw x5, 4(x9)              F D D E M W
E or x5, x5, x9               F F D E M W
```

(D stalls)

e needs x9 in E

Forwarding Pairs:

B → C C → D (C → E not fwd)

Exam 1 Review

9.25.25
L10

Exam 1 Topics:

- Instruction set architectures, RISC-V intro
  → NO allocate
- Performance
  - CPU Time, Amdahl's Law, etc...
- RISC-V Instructions and Programming
- RISC-V Instruction Formats
- RISC-V Procedures
- Pipelining
  → no control hazards

Ex. 5 stage RISC-V with forwarding

```
1. lw x5, 0(x4)        F D E M W
2. sw x7, 4(x5)          F D D E M W
3. add x9, x11, x7         F F D E M W
4. sub x7, x6, x9            F D E M W
```

a) How many instances of forwarding in this pipe trace?
note: an instruction whose source registers are from forwarding should be counted twice

1 → 2 is a load use hazard
fwd pairs: 1 → 2 (x5) } 2
3 → 4 (x9) }

b) For this pipe trace, how many cycles will this pipeline stall? 1

from the 1 load-use hazard

c) CPI = ideal CPI + stall CPI = $1 + \frac{1 \text{ stall}}{4 \text{ instr}} = 1.25$

SB Type:

blt x3, x1, LBL

↳ 4 bits after blt

SB imm[12|10:5] rs2 rs1 funct3 imm[4:1|11] opcode

0000000 00001 00011 100 10000 1100011

imm[13] = 0000000010000

* more exs of decoding in slides *

Amdahl's Law: optimize the common case

$$\text{Speedup} = \frac{1}{(1 - \text{Frac}_{enh}) + \text{Frac}_{enh}/\text{Speedup}_{enh}}$$

Fraction of the "problem" that we can improve (Frac_{enh})

Speedup of Frac_{enh}: Speedup_{enh}

CPU Time:

$$\text{CPU Time} = \text{instruction count} \times \text{cycles per instruction} \times \text{cycle time}$$

↓

weighted avg

10.2.25
L11

Control Hazards

Pipeline Hazards

- Structural: due to resource conflicts
- Data: due to instructions reading/writing data
- Control: due to branches

Branch Terminology

Branch Taken: condition met

Branch Not Taken: condition not met

* Control Hazard Example in Slides! *

```
    if (i < 5)
      .
      .
    else
      .
      .
      .

    loop: blt x10, x15, IF
          addi          <-- NT
          addi
          j DONE
    IF:   addi          <-- correct path begins here
          .
          .
```

But how does the pipeline know where to go?

Option 1: Sequential + Squash / Flush
(Q+RV5.in default)

1. Fetch instructions sequentially
2. Assume branch NOT taken
3. Squash signals when realize mistake

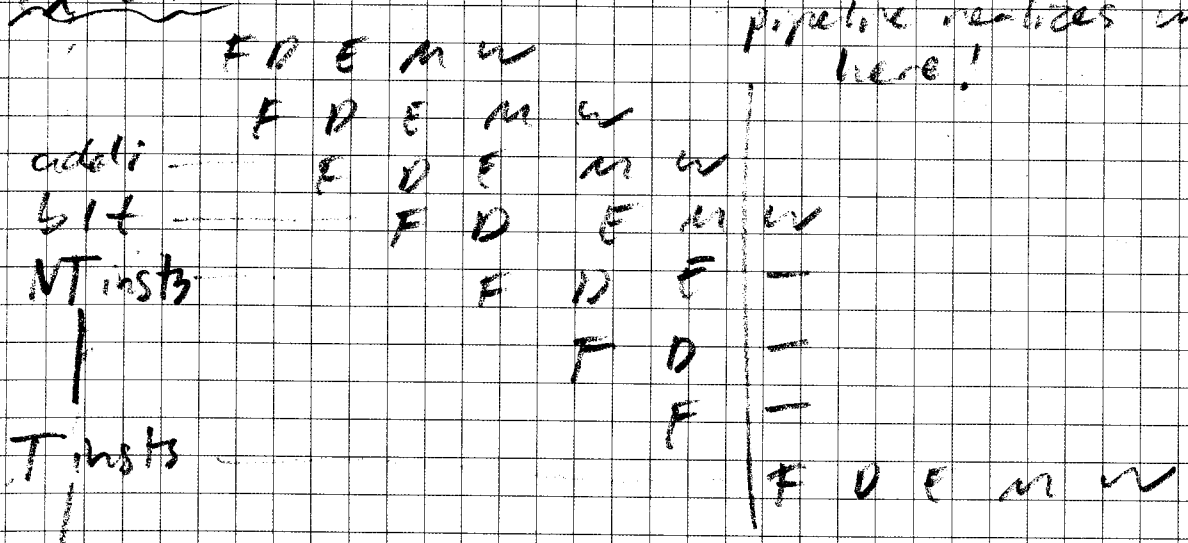
Squash → clear pipe registers

→ force signals to 0 to ensure no RF or Mem updates

→ In a deeper pipeline, RF and/or Mem may change and must be undone

→ This is called "flushing the pipeline"
may take additional time

Ex (cont...)



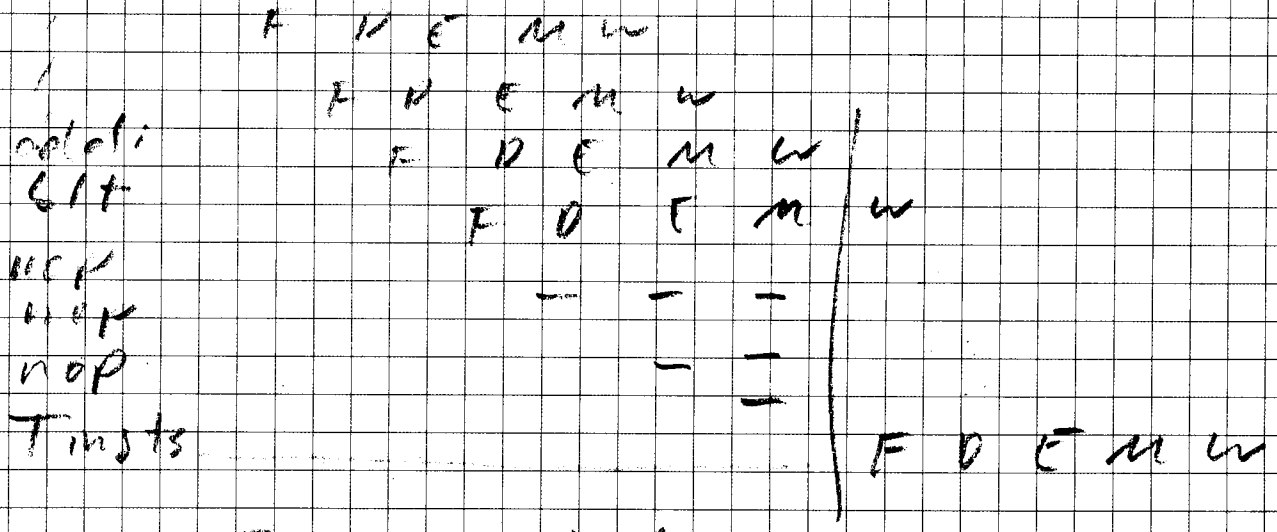
this is 3 stall cycles

and if branch not taken this would work out with no hazards!
no stalls!

Option 2: Stall after each Branch

1. Fetch instruction sequentially
2. Stall when branch encountered
3. Stall — do not fetch any instructions ("bubble fetch")

Ex cont... branch taken!



** Still 3 stall cycles! **

But now if branch NOT Taken...

- the pipe trace would look the same
- still 3 stall cycles
- just take NT insts instead of T insts at the end

Tradeoffs:

Option 2 simpler, but has more stall cycles if branch NT

Option 3: Branch Prediction

Branch Prediction: make an educated guess
deal with the fallout

* Options 1-2 are forms of "static" branch prediction

Problems.

in a standard pipeline the machine does not know
1. that an instruction is a branch until after D
2. which direction a branch is going until
after E (midway through M)
3. the target address of branches/jumps
until after E

Ex. jmp x0, 4 (x10) → PC becomes R[x10] + 4

Dynamic Branch Prediction:

hardware makes guesses on the fly

Ex. based on past history
(treated as a % for now...
is "predicted correctly 80% of the time")

* Ex: of performance impacts of this
in stalls

Control Hazard Ex 5

Performance Impact Ex:

Default Assumptions:

1. 5-stage (IFDEMW) with forwarding and WD
2. 1-cycle stall for load-use
3. How to treat control hazards will always be specified

Assume the following:

- 22% of instructions are loads
  - 50% of the time, the next instruction uses the loaded value
    - load-use hazard
- 13% stores
- 19% conditional branches
- 2% unconditional branches
- 44% ALU instructions

Also,

- 1 cycle stall when mispredict
- Branches are resolved in/branch at end of D
- 85% of conditional branches are predicted correctly

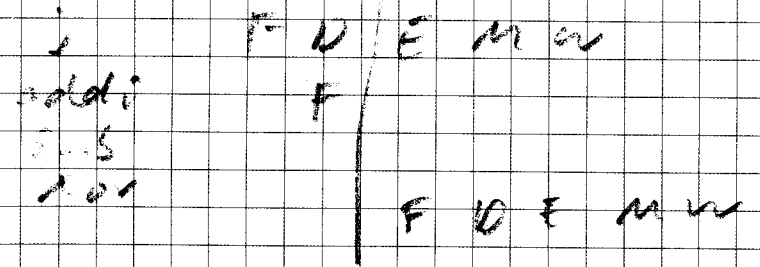
Cycle	1	2	3	4	5	6	7
Branch	F	D	E	M	W		
wrong path	F						
correct path		F	D	E	M	W	

Stall CPI. 0.23 × 0.5 × 1 cycle = 0.115
0.19 × 0.25 × 1 cycle = 0.0475
+ 0.01 × 1 cycle = 0.01
0.1825

Total CPI = 1.1825

ex. load-use hazard contributes to stalls
conditional branch mispredicts
unconditional branches

ex. j END
addi ...
sub ...
END: xor ...



unconditional branch cause 1 cycle stall

Ex. Pipe trace w/ Control Hazards

- Default assumptions
- predict beq NOT taken
- branches resolved after E
- 2 cycles spent flushing after branch E but before F of correct inst
- assume x2 = 0, x3 = 0

  branch SHOULD be taken

```
lw  x1, 4(x9)        F D E M W
add x4, x1, x9         F D D E M W  |
sub x2, x4, x9           F F D E M  | W
beq x2, x3, L                F D E  | M W
add x9, x8, x7                 F D  | - -
and x9, x5, x5                   F  | - -
L: add x4, x5, x9                   | - - F D E M W
                       (F D E M W)
                        ^ if correct prediction
                                          4 cycles of stall
```

Ex cont.

Branch predictor predicts taken!

```
lw      F D E M W     |
add       F D D E M W |
sub         F F D E M | W
beq             F D E | M W
add                   |
and                   |
L: add          F D   | E M W
```

Summary:

Default Assumptions:

1. 5 stage pipeline with forwarding
   → follow normal steps from before
2. 1 cycle stall for load-use
3. Control hazards will be specified
   → Read instructions carefully:
   a) what branch is supposed to do based on register state
   b) what branch actually does based on prediction

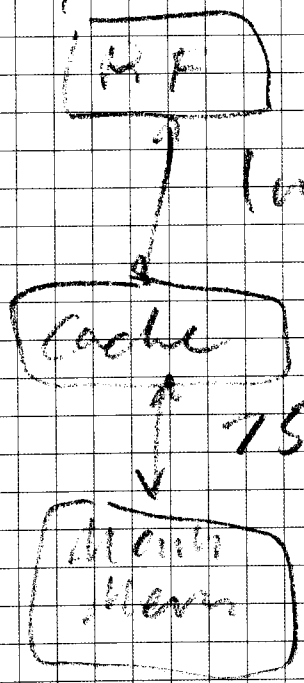
Pay attention to taken vs. not taken:
taken: PC = LBL
not taken: PC = PC + 4

Think about stall cycles vs "time to flush"
→ stall cycles are "gaps" in the pipeline caused by mispredict flush

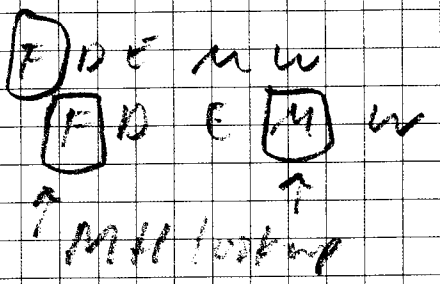
How important are caches? (1)

- CPU runs 3 GHz clock rate
- cache access time is 1 ns
- 90% of the time data is found in cache
- main memory access time is 75 ns
- 1/3 insts are loads/stores
- Base CPI w/o cache is 1
- No stalls from load-use or control hazards

Memory hierarchy:



cold
lw



What is the impact on the CPI?

instruction fetch loads/stores

$$1 + 0.1 * 75\text{ns} + 0.1 \times 75\text{ns} \times 0.333$$

↑ ideal/base CPI; ↑ miss rate; ↑ miss penalty

$$= 10.975 \text{ CPI}$$

Average Memory Access Time (AMAT):

standard metric for gauging performance of MM

AMAT = hit time + (miss rate * miss penalty)

↳ assumes 2 levels, for now

* Applies to entire Memory Hierarchy (MM)
for any amount of levels

from last ex,

AMAT = 1 + (0.1 * 75ns) = 8.5 ns

Direct-mapped and Associative Caches

10.9.25
43

Block/line: minimum amount of information transferred in memory hierarchy

Block Size: typically a power of 2, depends on architecture

RISC-V → 5B

Key Ex: How does a block go in the cache?

Block Addr (dec)	Addr (dec)	Main Mem (hex)
0	0	0x . . . 01
	4	0x . . . 0A
	8	0x . . . CD
1	12	0x . . . AD
	16	
2	20	
	24	
3	28	

lends

2 word transfer

Cache ex setup:
4 blocks, 2 words per block, 4 bytes per word

index	1			0				word byte
	3	2	1	0	3	2	1	0	
0	00	00	00	CA	00	00	00	01	
1	FE	ED	AD	AD	AB	CD	AB	CD	
2									
3	**					0			

note. $K_i = 2^{10} = 1024$ # use this #
$K = 10^3 = 1000$

Q: $N \frac{\text{bytes}}{\text{cache}} = \frac{4 \text{ blocks}}{\text{cache}} \times \frac{2 \text{ words}}{\text{block}} \times \frac{4 \text{ bytes}}{\text{word}} = 32 \frac{\text{bytes}}{\text{cache}}$

Q. In a direct mapped cache, an address maps to the cache **index** specified by:

Block addr % # blocks in cache

4 lw's

Addr (dec)	index
0	0 % 4 = 0
12	1 % 4 = 1
24	3 % 4 = 3
32	4 % 4 = 0

will evict block 0 (overwrite it) (not shown)

Memory Address → Cache Lookup

Prev ex used block addr, which is not really used in practice because:

→ How do we select an individual byte from a addr in mem (for lb)?

→ How do we differentiate between block addrs that map to the same cache block?

→ How do we determine block addr when given a mem addr?

Answer: Break up the mem addr

- byte offset ← Now since RISC-V has it
- block offset
- index
- tag

} Logic Design

Leftover bits

32-bit mem addr

Byte offset: Select one of 4 bytes in a word

RV32I = 2-bit Byte offset

block offset: select one of 2 words in block

index: select one of 4 indices in cache

Tag: Difference to differentiate addresses

→ addr 3 2 tag index block offset byte offset

0 0 0 1 0 0 0 0

Byte offset and Block offset:

Byte offset: set aside M bits to select individual byte in a word

RISC-V → M = 2 bits
RV64I → M = 3 bits

Block offset: set aside N bits to select individual word in a block

2 words per block → N = 1 bit
8 words per block → N = 3 bits

N = log(# words / block) → log2(# words / block)

Block Address and Index:

In Dinero, with M = 2 and N = 1

block address = 29 bits
index = 25 bits (bits 3 – 4)
15 = log(# blocks in $) = 27 bits

Tag = leftover bits
# bit addr – everything else

Design Decisions and Tradeoffs:

- Increased block size exploits spacial locality
- Increased cache size exploits temporal locality

We must make tradeoffs to decide between these two

Associative Caches and Exs

10.14.25
L14

N-way set associativity: each block maps to N cache locations, called ways

(1-set comprises N ways)

Block Addr (dec)	Addr (dec)	Main Mem (hex)
0	0	0x00000001
	4	0x0000000A
1	8	0xABCDABCD
	12	0xFEED0AD
2	16	0x1234567
	20	0x7654321
3	24	0x01010101
	28	0x18421842

Cache Ex: 4 blocks, 2 words per block, 4 bytes per word, 2-way set associative

Set (index)	Way	2				0				Word Byte
		3	2	1	0	3	2	1	0	
0	0	00	00	00	0A	00	00	00	01	
0	1									
1	0	FE	ED	AD	0D	AB	CD	AB	CD	
1	1	18	42	18	42	01	01	01	01	

note. (Block addr) % (# blocks in cache) no longer works!

need to determine the set to which a block address maps!

(Block address) %, (# sets)

sequence of loads:

Block Addr	Set
0	0
1	1
3	1

no collision since empty way available!

Replacement Policies:

- least recently used
- least frequently used
- random

Memory Addresses → Cache Lookup

Break up memory address:

- byte offset
- block offset
- Index → only to determine set
- Tag → still leftovers

} same calculations as before

31	...	4	3	2	1	0
Tag		Index	Block offset		Byte offset	

bits index = lg(# sets in $)

Cache lookup examples: more in slides

- Consider a 32-bit byte-addressable memory with 4-word blocks and a 128 KiB direct mapped cache. Determine bits for byte offset, block offset, index, and tag:

2^2 bytes/word, 2^2 words/block → 2^4 bytes/block

2^{17} bytes/cache

... see slides

Cache simulator ex in slides too!

note. Miss cases

1. Empty
2. Tag Mismatch

Cache Performance

10.16.25
L15

Note: writes = stores
reads = loads

What About Writes?

4 ex in slides

Addr 0x00000008 wr data 0xFFFFFFFF

...0 0 0/0 1/0/0 0 → index = 1, tag = 0
T I B1 B0 B2 B0

hit because valid & tags match

Addr 0x0000002C wr data 0xEEEEEEEE

... 0 0 0 0 1/0 1/1/0 0 → index = 1, tag = 1
T I B1 B0 B2 B0

miss because valid but tag mismatch

1) Evict I = 1, T = 0
2) Load I 1 T 1 to the cache
3) update cache w/ 0xEEE...

Key Question for Writes?

When do we update memory after a store?

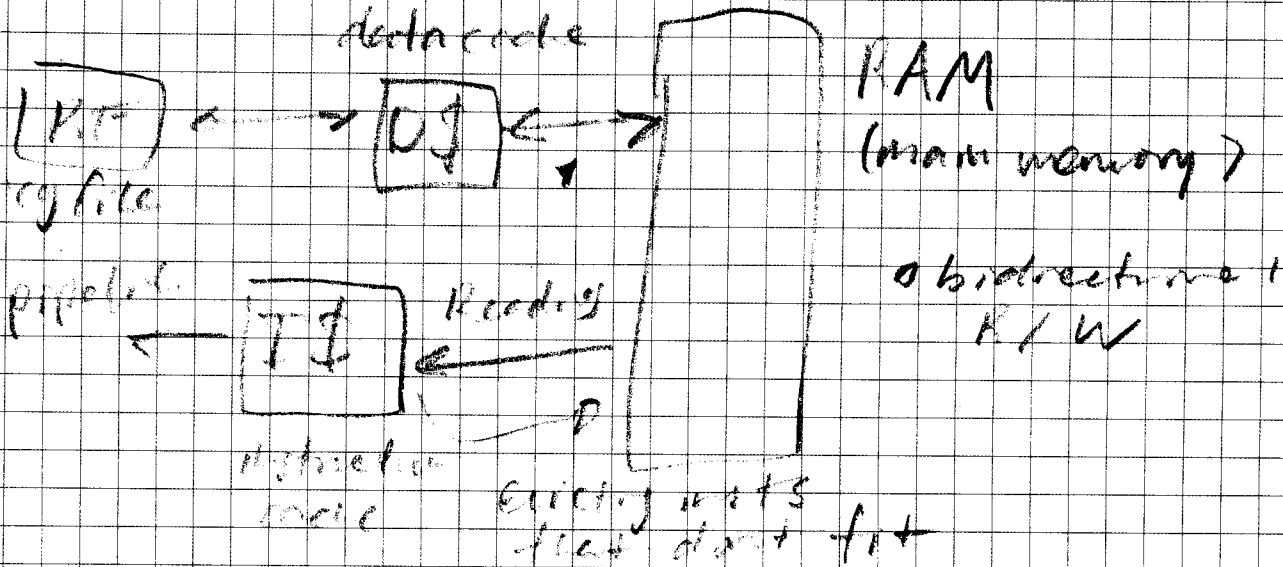
write-through $: write memory immediately

write-back $: write memory when block is evicted

Pros:
- only send "last" update
- simple

Cons:
- waste (frequent stores)
- tracking

Split Caches:



Improving Performance:

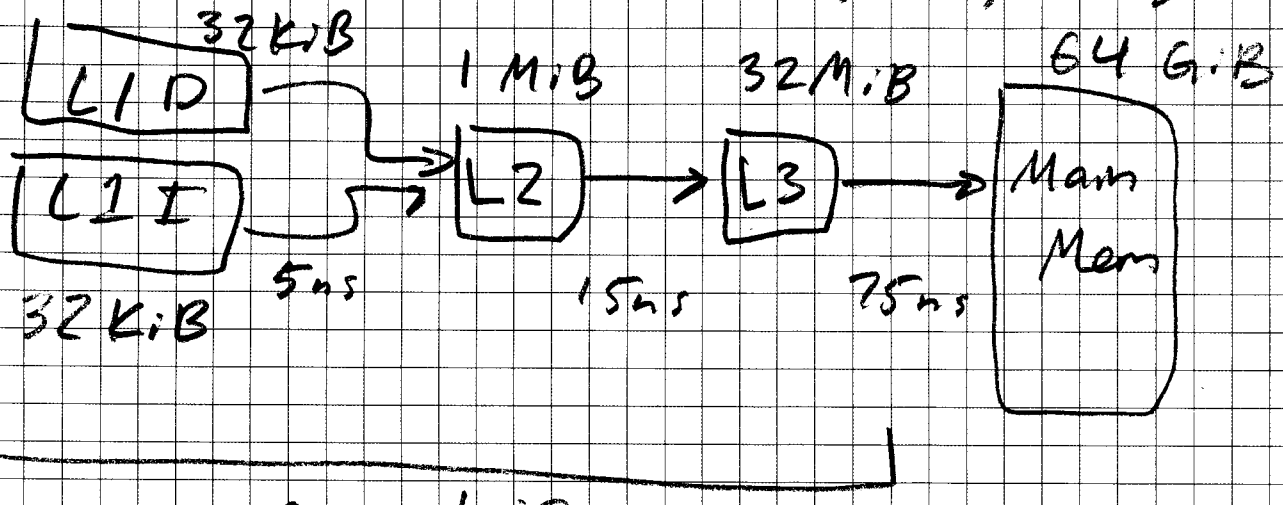
Average Memory Access Time =

Hit Time + (Miss Rate × Miss Penalty)

↓ tradeoff ↓ ↓

- Smaller ↓ ↔ • bigger ↓ • Interconnect
- pipelined ↓ • more words per block (block size ↑) • Improve memory
  • more blocks (index ↑)

Multi-level caches: modern machines typically have 3 levels of cache (L1, L2, L3)



What portion of AMAT does this address?

More on-chip storage → lowers miss rate
L2 → lowers hit time
→ lowers miss penalty

L1 case: $AMAT = HT_{L1} + (MR_{L1} \times MP_{L1})$ ← original formula

L2 case: $AMAT = HT_{L1} + (MR_{L1} \times (HT_{L2} + (MR_{L2} \times MP_{L2})))$

L3 case: $AMAT = HT_{L1} + (MR_{L1} \times (HT_{L2} + (MR_{L2} \times (HT_{L3} + (MR_{L3} \times MP_{L3})))))$

Reducing Miss Rate:

Compulsory miss: "cold-start miss"
A cache miss caused by the first access
to a block that has never been in the cache

Capacity miss: The cache, even with ideal full
associativity, cannot contain all the blocks
needed to satisfy the request

Conflict miss: "collision miss"
Occurs in set-associative or direct-mapped caches
when multiple blocks compete for the
same set that would be eliminated in
a fully associative cache of the same size

Strategies:

Compulsory → prefetch (prediction)
→ (increase block size)

Capacity → Make $ larger

Conflict → increase associativity
→ Make $ larger
→ (increase block size)

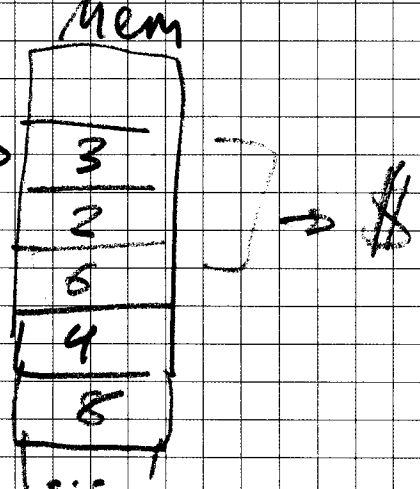
Virtual Memory

10.28.25
L16

Cache-Aware Programming: Loop Interchange

Recall: how are 2D C arrays stored in memory?

$$arr = \begin{bmatrix} 3 & 2 & 6 \\ 4 & 8 & 15 \\ 16 & 23 & 42 \end{bmatrix}$$



this is called row-major ordering!

```
int array[100][50];
for (int i=0; i<100; i++)
    for (int j=0; j<50; j++)
        arr[i][j] = i + j;
```

interchangeable

* this ordering takes advantage of spacial locality better!
  - compiler does this automatically sometimes

more examples of this in slides

LRU vs NRU vs RRIP:

all replacement policies for caches

details in slides

Virtual Memory (VM):

LARGE overlap with OS!

arch → efficiency
OS → control

Motivations:

1. Allow processes to use more memory (RAM) than exists on the system
2. Simplify programming model
3. Enable protection among processes

Terminology:

Physical memory: actual memory (RAM) in the system

Virtual memory: illusion of large virtual address space
- allows the hard drive to be used as a backing store for main memory!

Translation: mapping virtual addresses to physical addresses

OS and hardware work together to manage translation

Pages:

treat memory as a set of pages
- page sizes are typically very large (determined by HW and OS)

4KiB in this class!

Ex. 4KiB page size on 32-bit system

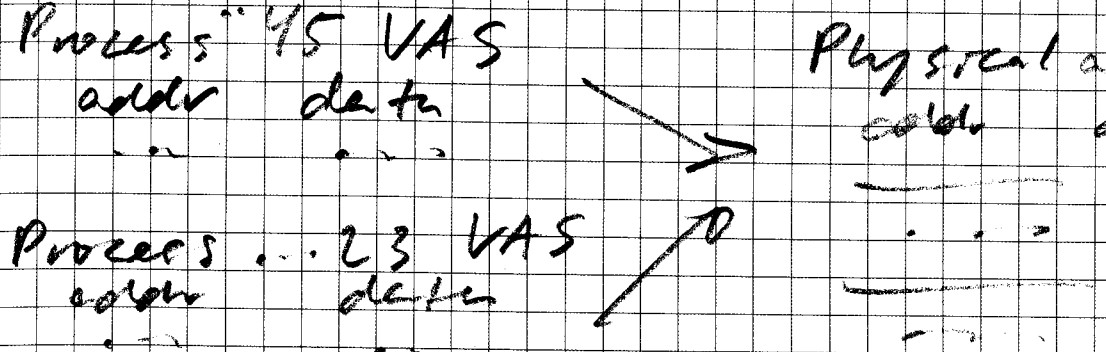
4096 bytes/page = 2^{12} B/page

How many pages in system?

2^{32} bytes / 2^{12} bytes/page = 2^{20} pages

	Addr (dec)
page 0	0
	1
	...
page 1	4095
	4096
	...
page 2	8191
	8192
	...
page 3	12287
	12288
	...
page 4	16383
	16384
	...

Address Space: each process sees a contiguous virtual address space



Virtual Memory (Paging)

10.30.25
L18

Address Translation:

1. Determine the virtual page number
   → upper bits of the virtual address
2. Look up the physical frame number in a per-process structure called a page table
3. Complete the address: physical frame number + lower bits of VA

Part 1: VPN, offset

Consider process A, 32-bit system, 4KiB pages

2^{12} bytes

Addr (dec)

page 0 { 0 ... 4095
page 1 { 4096 ... 8191
page 2 { 8192 ... 12287
... 12288

When process A executes:
lw x5, 0(x15) // x15 = 8192,
x5 = M[8192]

VPN = 2

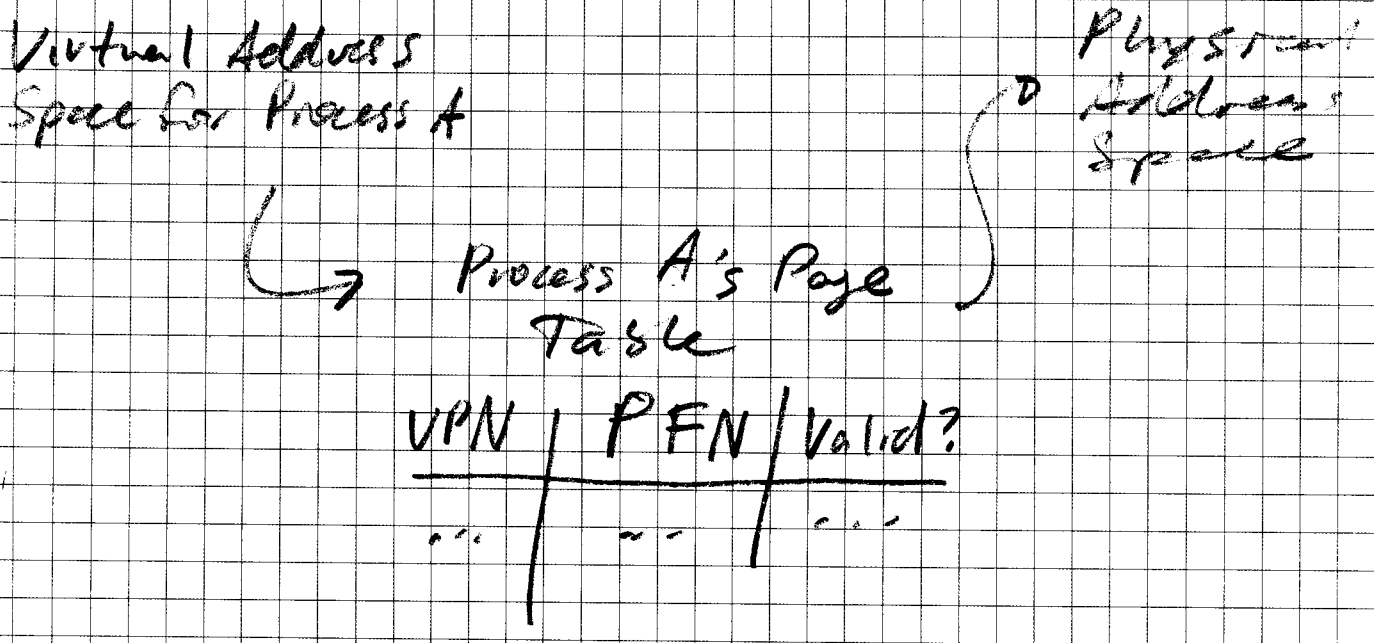
More formally,
2^{32} bytes · (1 page / 2^{12} bytes)
$= 2^{20}$ pages → 20-bit VPN

For virtual address 8192 =
0x00002000

VPN = 00002, offset = 000

What is the purpose of the offset? Identify the byte within the page

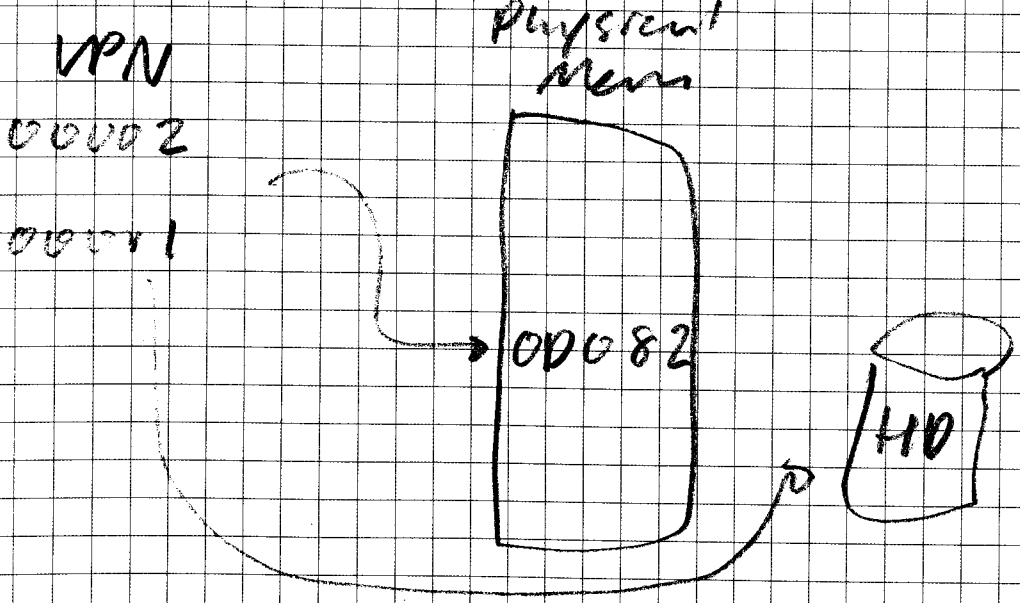
Part 2: Physical Address



Physical Address = PFN + offset

- PFN: from page table
- concatenation
- offset: from virtual address

What about the hard drive?



Page fault: page table indicates the page is on the hard drive (valid = 0)

- OS finds the page on from hard drive to physical memory
- Updates the page table with the new mapping
- Evicts a page if needed (based on some replacement policy)

Ex. 52-bit virtual address space
52-bit physical address space
16 KiB page size $\rightarrow 2^{14}$ bytes

1) How many virtual pages?

$2^{52} B / 2^{14} \frac{B}{page} = 2^{38}$ virtual pages

VPN size? 38 bits

2) How many physical frames?

$2^{42} B / 2^{14} \frac{B}{page} = 2^{28}$ physical frames

PFN size? 28 bits

3) How many bits offset? 14 bits.

Virtual Address: 1ABCDEF9

VPN: isolate leftmost 38 bits

offset: isolate rightmost 14 bits

Physical Address:

offset is the same!

PFN: leftmost 28 bits

Page Table Organization: page tables are designed to avoid conflicts that lead to page faults

page fault: requested virtual page not mapped in physical memory

- On Disk
- Not allocated

Memory Hierarchy Now...

L1$	L2$	L3$	Memory	Disk
1ns	15ns	30ns	100ns	100ms

Page Table:

VPN	PFN	Valid?	Dirty?	LRU?
...	...	...	...	...

Any PFN can be mapped to any VPN

note. Typically, all pages are mirrored on hard drive to avoid unnecessary transfers

"dirty" bit tracks pages that have been written, hard drive copy must be updated

VM Performance (TLB)

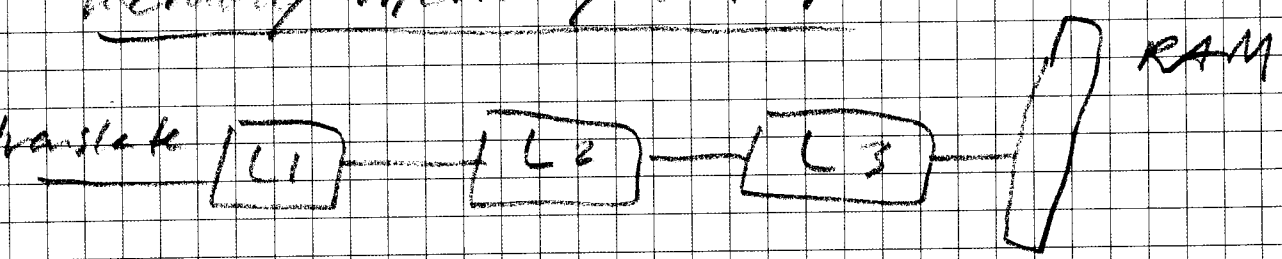
11.4.25
L19

Page Tables: stored in mem hierarchy
→ many challenges

1. Page Tables are large:
   if each page table entry (row) is 4 B
   (2^{20} page table entries) · (2^2 B / entry)
   = 2^{22} B = 4 MB page table / process

2. Caches store physical addresses:
   must translate first — slow!

3. Every memory access requires two
   memory hierarchy lookups



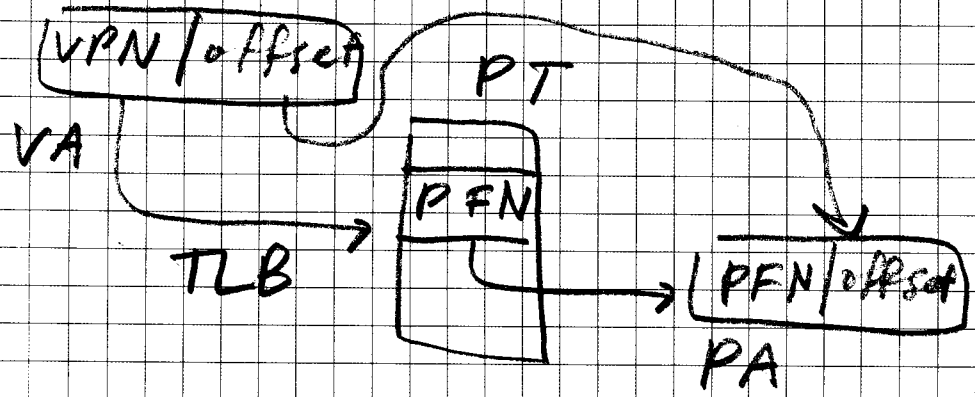
Solution: special cache structure called
translation lookaside buffer (TLB)

lw → translate
   → load access from RAM

TLB: translation cache

Translation with TLB:

Recall: translation now required from virtual → physical address



TLB: Translation Lookaside Buffer

- small, fast, fully associative → no set index
- VPN is used as a tag, if present, can be translated immediately

Ex. Process A's Page Table:

VPN	PFN	Valid?

TLB:

Valid	LRU	Tag (VPN)	PFN (Data)

Translation with TLB - Entire Process

Ex. Fully TLB, 4KB page size, 32-bit VAS, 28-bit PAS

4 KB page size → 2^{12} B → 12-bit offset

VA | VPN | offset
31 12 11 0

PA | PFN | offset
27 12 11 0

→ 20-bit VPN, 16-bit PFN

TLB

Valid	LRU	VPN	PFN
:	2 1 0 3	00003	ABCD
:	3 2 1 0	ABCDE	DCBE
:	1 2 1 0	00002	1F26
0	0 3 2 1	00004	0096

Process A Page Table

VPN	PFN	Valid
00000	0000	1
00001	0A3K	0
00002	0052	1
00003	ABCD	0
00004	0096	1
...	...	...
FFFFF	0432	1

The following occurs on every (virtual) load:

0 0 0 0 2 | 0 1 0 — 1) TLB lookup

0 0 0 0 4 | A F — 2) TLB hit, so construct PA

0 0 0 0 3 | A 0 8 → 0 0 8 2 0 1 0 ← PA

VPN ↑ offset ↑

Ex cont:

VPN	offset
00004	1AF
00005	408

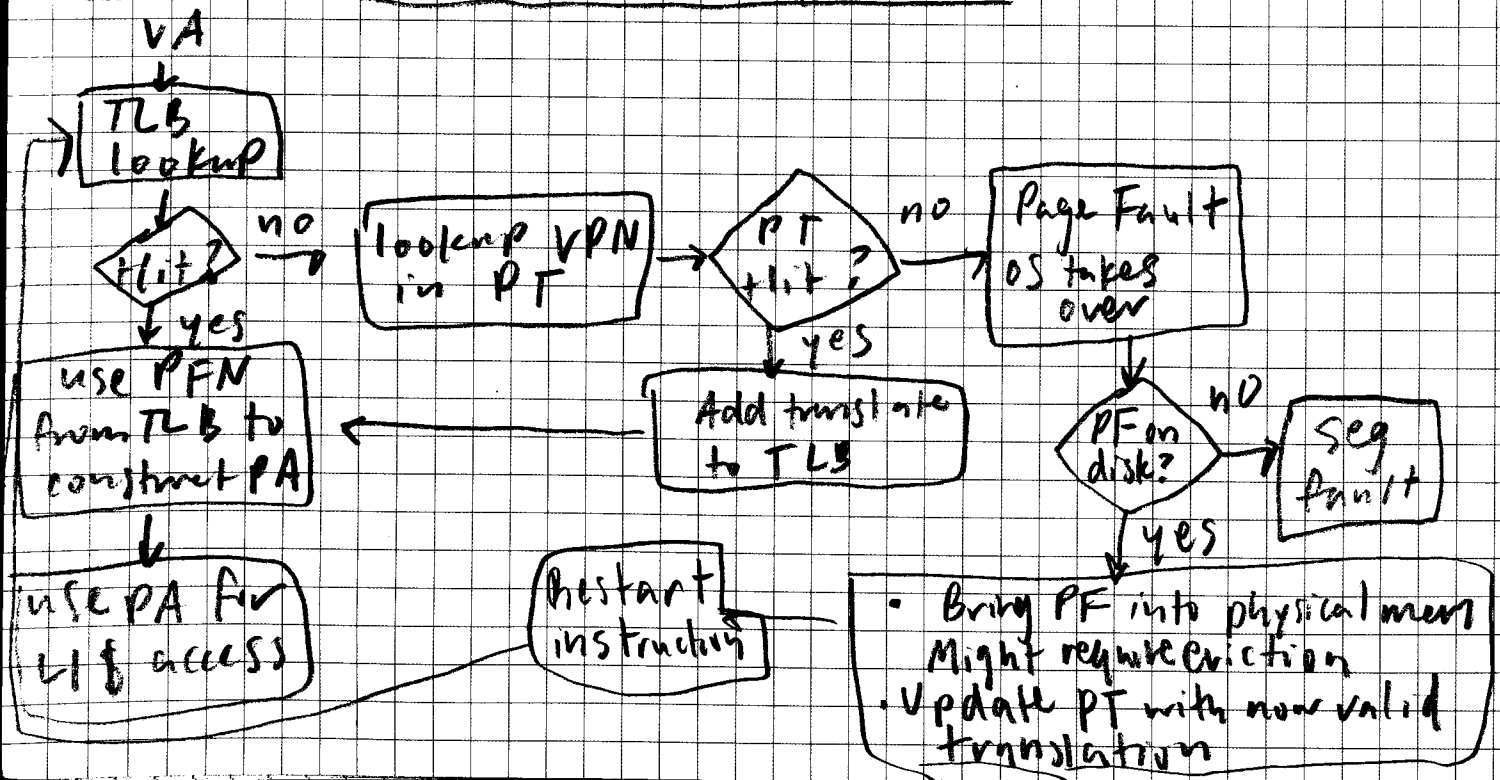
00002 | 010

1) TLB lookup
2) TLB hit, so construct PA
3) PA: D08201D

1) TLB lookup
2) TLB miss
3) P.T. lookup, PT hit (V=1)
4) update TLB (translate, LRU, V)
5) PA: 00961AF

1) TLB lookup
2) TLB miss
3) PT lookup, PT miss (V=0)
4) update PT
5) PT lookup
6) update TLB
7) PA: ABCDA08

Critical Path for a Memory Access:



11/6/25
L20

VM Wrap-Up and Static Scheduling

Complete VM Ex

2^4 Bytes / page

12-bit virtual address, 16 B page size

2^{12} addressable bytes in virtual address space

2^8 virtual pages

(2^{12} B / 2^4 B/page)

8-bit virtual page number (VPN):

lg(2^8 pages)

4-bit offset:

lg(2^4 B/page)

2^7 B

Physical memory size: 128 bytes

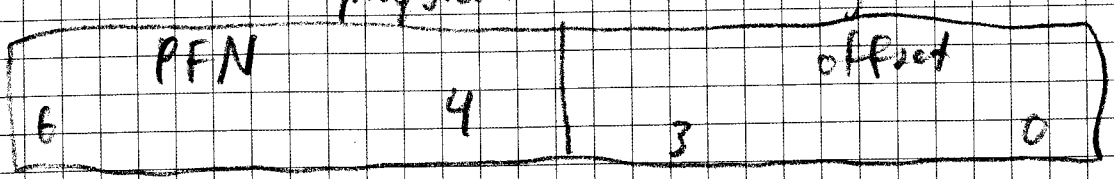
7-bit physical address:

3-bit physical frame number (PFN)

$\lg 2^3$ frames

virtual address bit ranges

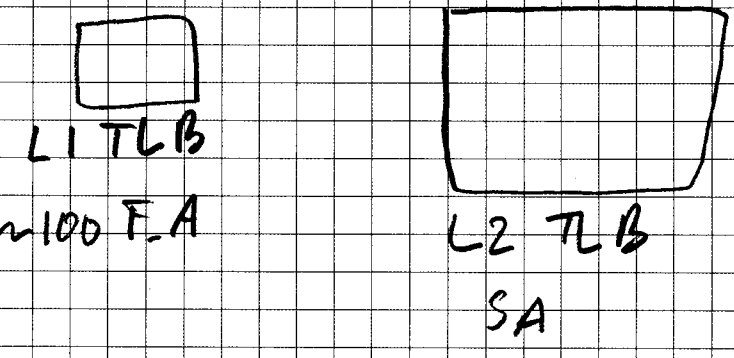
physical address bit ranges



Additional VM Optimizations:

TLB is a cache; all the same optimizations may be used

common: multi-level TLBs



translation process (including TLB), must support OS paging schemes

Replacement Policy Exs:

LRU is expensive, not used for TLB. instead LFU or Pseudo-LRU

- 4 entries in structure: could be 4-way SA cache, FA cache, TLB, page table, etc.
- LFU: assume a 2-bit saturating counter
- order of access in letter by 4-digit tag: A B C C D A E

LRU: M M M H M H M
LFU: M M M H M H M
PLRU: M M M H A H M

LRU = 0 → MRU = 3

Tag	LRU
A	0 3 2 1
B E	1 0 3 2
C	2 1 0 3
D	3 2 1 0

4 ways * 2B/way = 8B

0-3

Tag	LFU	
A	0 1 1 1	2
B E	0 1 1 1	1
C	0 1 2 2	2
D	0 0 0 1	1

Tag	PLRU	PLRU
A	0↑ 1↓	0↑ 1↓
C	0↑ 1↓	0↑ 1↓
B E	0↑ 1↓	0↑ 1↓
D	0↑ 1↓	0↑

0↑ 1↓ 3B total!

11.11.25
L21

Static Scheduling, Branch Prediction

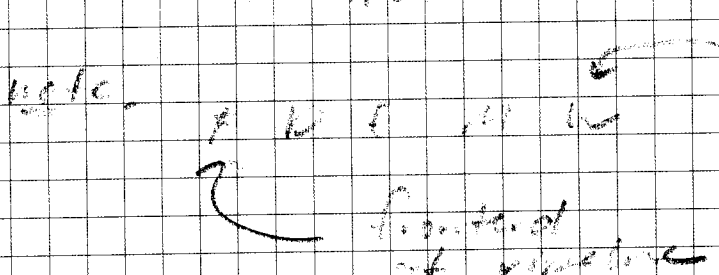
Toward an Out-of-Order Pipeline

The maximum IPC in our 5-stage pipeline is: **1**

Single-issue: send 1 instruction to a functional unit (eg. adder to the ALU)

Multiple-issue: send N instructions to functional units (N adders, Multipliers, etc.)

Ex. allow a pair of (ALU/branch) + (load/store)



Ex.

A ALU or branch inst. F D E M W

B load or store inst. F D E M W

A F D E M W

B F D E M W

A F D E M W

B F D E M W

maximum IPC for this pipeline: **2**

Static Scheduling:

Scheduling: deciding how to issue instructions

static scheduling: provide additional issue capabilities, software does scheduling

- compiler responsible for finding parallelism among instrs
- compiler responsible for avoiding or eliminating hazards

n.t.c. A = ALU or branch inst
B = load or store inst

Ex. Loop: lw x31, 0(x20) ← constant
add x31, x31, x21
sw x31, 0(x20) ← 4(x20)
addi x20, x20, -4
blt x22, x20, Loop

2 word; read/use record

Scheduled code:

A	nop	F	D	E	M	W	
B	lw	F	D	E	M	W	
A	addi		F	D	E	M	W
B	nop		F	D	E	M	W
A	add			F	D	E	M W
B	nop			F	D	E	M W
A	blt				F	D	E M W
B	sw				F	D	E M W

nop - "no operation"

Drawbacks and (Partial) Solutions:

In previous:

- 3 nops inserted, not good use of resources!
- Pipeline has capability of 2 IPC (cores), but we only achieved $5/4 = 1.25$ IPC

Partial Solution, loop unrolling: compiler makes multiple copies of loop body

use extra instr to fill multiple-issue pipeline!

(details on Friday)

Static Scheduling Summary:

Benefits: can yield improved IPC

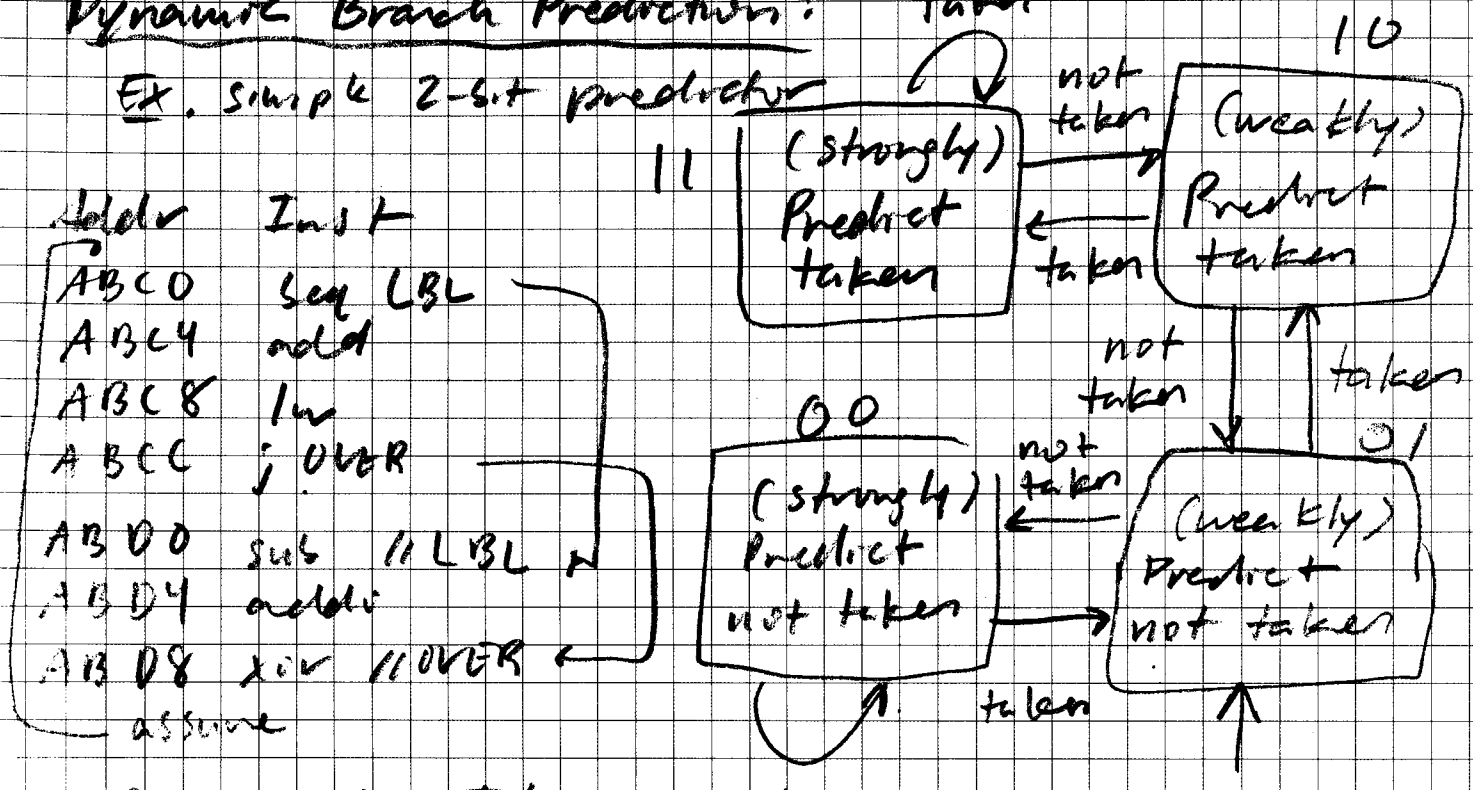
loop unrolling can yield additional improvements

Challenges:

- Significant amount of dependency checking
- loop unrolling leads to a larger code footprint
- unrolling across branches requires more code to handle misspeculations
- limited register resources for unrolling

Dynamic Branch Prediction:

Ex. simple 2-bit predictor



Addr	Inst
ABC0	Seq LBL
ABC4	nop
ABC8	lw
ABCC	j OVER
ABD0	sub // LBL
ABD4	nop
ABD8	xor // OVER

assume

Branch History Table

opcode[3:0]	Prediction?
0	~~0r 00 or~~ 10
4	01
6	01
C	~~or to~~ 11

Branch Target Buffer

PC	Target?
ABC0	LBL (ABD0)

Exam 02 Review

11.13.25
L22

HW08)

20-bit VA
18-bit PA
64 KiB page size = $2^6 \times 2^{10}$ B = 2^{16} B
↳ 16 offset

First Access: 0x34815

VPN	offset
19 16	15 0

- VPN 3 not in TLB (miss)
- PT says VPN 3 is valid
  - maps to PFN 2
- update TLB with new translation

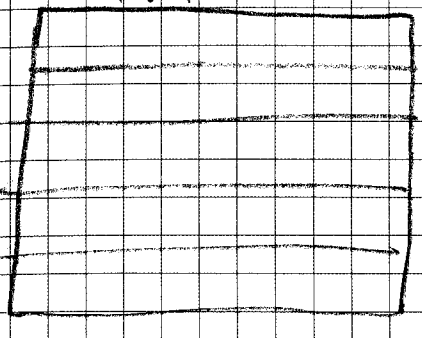
V	PID	VPN	PFN
1	2 (ASID)	3	2

P.A (2 bit PFN) = 34815

PFN	offset
19 16	15 0

Fully associative $: any block can map anywhere → no conflicts

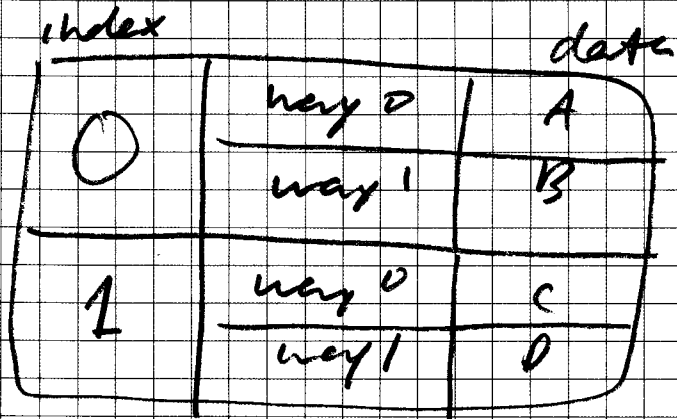
FA



DM

all accesses may map to the same index

N-way set associative:



less cache misses, but must search through all ways to find data — so longer access time!

AMAT.

1-level: $AMAT = HT + (MR \times MP)$

2-level: $AMAT = HT_{L1} + (MR_{L1} \times MP_{L1})$

$MP_{L1} = HT_{L2} + (MR_{L2} \times MP_{L2})$

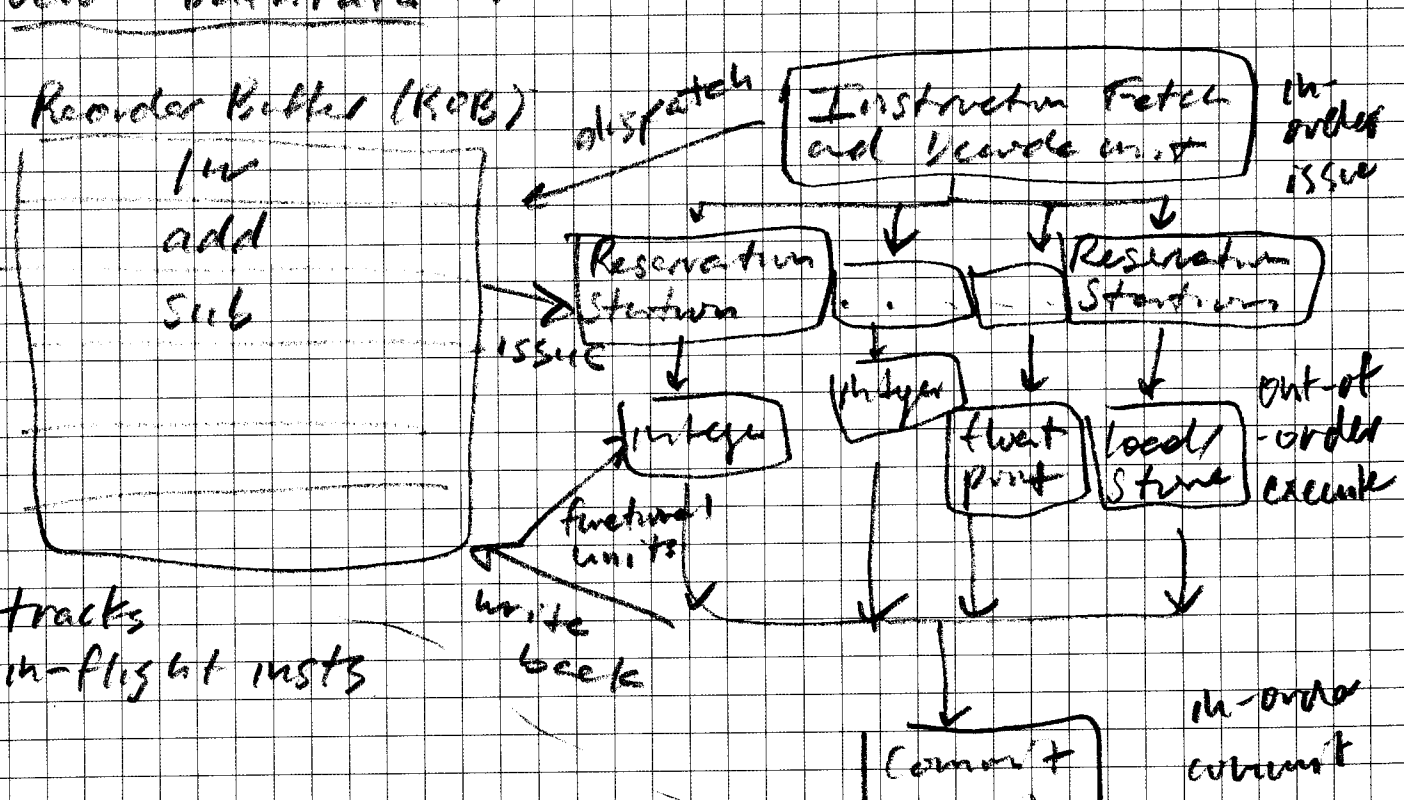
Dynamic Scheduling

11.20.25
L24

Scheduling: deciding how to issue instructions

Dynamic Scheduling: pipeline issue capabilities, hardware does scheduling

New "DataPath":



5-stage in order: FDEMW

Dynamic: F D I E M W C

New Stages: frontend E M

Backend

1. Fetch
2. Decode + Dispatch (send inst to ROB)
3. Issue (send inst to reservation stations)
4. Execute
5. WriteBack (write to ROB, NOT RF or Mem)
6. Commit
   inst at head of ROB → time to write to RF/Mem remove from ROB

OOO: New Dependences and Hazards

Antidependency: Write-After-Read (WAR) hazard
(also called name dependency)

```
add  x10, x4, x9     read
sub  x4, x12, x5     write
or   x15, x4, x3
```

no relationship b/w add/sub, but if sub executes before add → WAR

Output dependency → Write-After-Write (WAW) hazard

```
srl  x10, x29, x19   write z
add  x10, x3, x10    write y
and  x2, x10, x3
```

no relationship b/w srl/add, but if add executes before srl → WAW

Solution in both cases: register renaming

Recall: True dependency

```
lw   x10, 0(x5)      write
add  x11, x10, x8    read
```

Register Renaming:

Architectural registers (x0–x31) are **names**, mapped to **physical** register locations

- create a large pool of physical registers called "tokens"

  t0, t1, t2, ..., tN

  key new structure B

- look up tokens in **map table**
- every destination register is renamed

(RAT)

Register Map Table

x1	t10
x2	t11
x5	t12
x4	~~t13~~ t101
x5	t14
x6	t15
x7	t16
x8	t17 t102
x9	t18
x10	~~t19~~ t100
...	

Addressing WAR:

add x10, x4, x9 → add t100, t13, t18 → ROB

sub x4, x1, x5 → sub t101, t10, t14

or x6, x4, x2 → or t102, t101, t12

Address WAW:

same process!

sub x10, x24, x19

→ sub tF, ..., ...

add x10, x3, x10 → add tG, ..., ...

and x2, x10, x3 → and ..., tG, ...

Free List: t60, t61, t62, ...

key new structure C

11/25/25
L.25

Dynamic Scheduling cont.

Key Questions

1. How can (does?) hardware "detect" a hazard?
   instruction whose operands aren't ready cannot leave reservation stations
2. How do we allow "later" (subs) instructions in stream to execute first (before add)?
   instructions whose operands are ready can leave reservation stations
3. How do we ensure program order and memory is maintained?
   in-order front end and in-order commit - register renaming to eliminate WAR and WAW

Example #1:

Assumptions:

- pipeline can issue 2 insts per cycle of any type
- all dest regs renamed in D
- each inst spends at least 1 cycle in reservation station
- each ALU inst takes 1 cc in E
- branches resolve in E and stomp on
- insts must wait in reservation stations in produced Rs, and execute in next CC

Code:

```
lw  x1, 0(x10)      (lw in E 5 CC's)
cold x5, x1, x2     # misses data cache, 5 CC stall
 sus  x1, x9, x4
beq  x5, x1, A      # will always predict NT
or   x5, x6, x7
addi x5, x5, 1
 j   B
 i
A:  and x5, x6, x7
B:
```

misses instruction $, 20 CC stall

□ functional units

reservation stations

□ : started in ROB, sent to RS

Register Map Table

Reorder & Reorder Buffer

Dispatch

Inst F & D

Issue

10	lw t30, 0(t20)				
12	cold t31	t30, t21			
12	sus t32	t28, t23			
13	beq t31, t32, A				
13	or t33	t25, t26			
13	addi t34	t33, 1			

WB

Free List: t50, t51, t52 ... t50, t20, t22 t30 t33, t34

wt = waiting after W to commit

R = waiting in reservation station

commit unit

Pipe Stages: F, D, I, E, W, C

```
lw    t30, 0(t20)
cold  t31, t30, t21
sus   t32, t28, t23
beq   t31, t32, A
or    t33, t25, t26
addi  t34, t33, 1
 j    B
 i
A: and
B:
```

F	D	I	E	E	E	E	E	W	C
F	D	I	R	R	R	R	R	E	W C
	F	D	I	E	W	wt	wt	wt	wt W C
	F	D	I	R	R	R	R	R	R E C
		F	D	I	R	R	wt	wt	wt wt wt wt
		F	D	I	R	R	E	W	wt wt wt wt
			F	F	F	F	F	F	F F F
			F	F	F	F	F	F	F F F

wrong path

12.2.25
L.26 Dynamic Scheduling and Parallelism

Example #2: Pipe Trace

Free List: t50, t10, t75, t16, t99, t20, t100, t5, t15, t25

Assumptions (6) (8) (13) (13) (14) (19) (15)

- pipeline can issue 2 insts per cycle of any type
- all destination registers renamed in D
- registers returned to free list in C
- each inst spends ≥ 1 CC in reservation station (I)
  - All ALU units take 1 CC in E except mul which takes 4 CC
- Branches resolve in E and stop W
- insts wait in reservation stations in between I, W, and E in next CC
- unlimited ROB, RS, and Srs renames, and 8 ALUs

Pipe Stages: F D I E W C

(for ldst, E includes mem access)

Inst	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
add x1, x1, x1	F	D	I	E	W	C										
① t50, t15, t15																
add x1, x1, x1	F	D	I	R	R	E	W	C								
t10, t50, t50																
mul x1, x1, x1		F	D	I	R	R	R	E	E	E	E	W	C			
t75, t10, t10																
sub x2, x2, x2		F	D	I	E	W	Wt	Wt	Wt	Wt	Wt	Wt	C			
t16, t17, t17																
add x1, x2, x2			F	D	I	R	E	W	Wt	Wt	Wt	Wt	Wt	C		
t99, t16, t16																
mul x2, x3, x3			F	D	I	E	E	E	E	W	Wt	Wt	Wt	C		
t20, t25, t25																
add x1, x1, x1				F	D	I	R	R	E	W	Wt	Wt	Wt	Wt	C	
t100, t99, t99																

- default assumptions for most of these questions

Register Map Table

x1	t15, t50, t10, t75, t99, t100
x2	t10, t16, t20
x3	t25
x4	t19
x5	t90
x6	t55
x7	t89
x8	t8
x9	t26
x10	t4
...	

CO removed | Reorder Buffer (ROB)

6	add t50, t15, t15
8	add t10, t50, t50
13	mul t75, t10, t10
13	sub t16, t12, t17
14	add t99, t16, t16
14	mul t20, t25, t25
15	add t100, t99, t99

① write, but data cannot persist even after renaming

note: when returning registers to the free list, return the previous register to the current destination register from the RMT

Types of Parallelism:

Flynn's Taxonomy:

Instruction Streams	Data Streams: Single	Data Streams: Multiple
Single	✓	
Multiple		

✓ everything we've covered in this class so far have been in this quadrant

... but other architectures exist!

Cache Coherence

12.4.25
L.27

Types of (Hardware) Multi-threading:

Thread: lightweight process

Goal: add additional threads use (underutilized) resources

SMT: Simultaneous multithreading

Three Challenges in Exploiting Parallelism:

1. Making efficient use of multiple threads, cores, sockets, etc
2. Storing memory effectively
   - NUMA and latency
   - Bandwidth
3. Coherence (and consistence)

Challenge 1: Efficient Use of Resources

$$\text{Speedup} = \frac{1}{(1 - \text{Fraction}_{\text{parallelizable}}) + \frac{\text{Fraction}_{\text{parallelizable}}}{N}}$$

- performance improvements dictated by Amdahl's Law

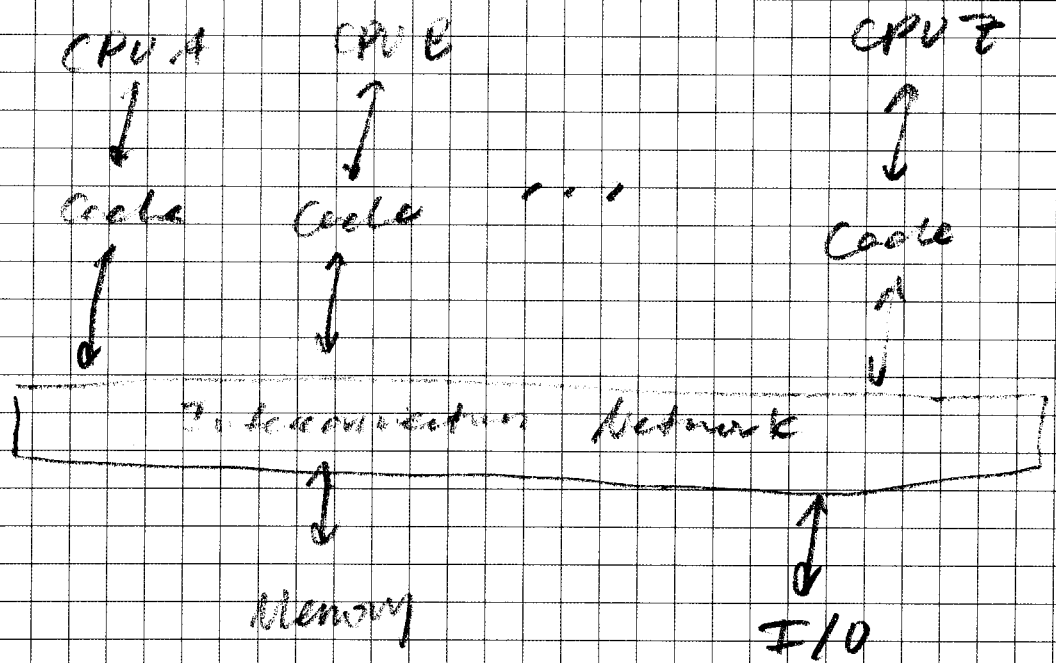
Challenge 2: Sharing memory effectively

- centralized shared memory
- distributed shared memory

& each has drawbacks, out of scope for this course

Challenge 3: Coherence (and Consistency)

Many threads, one address space, shared memory, write-back caches



Time Step	Event	Cache A	Cache B	Mem
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A stores 1 into X	1	0	1

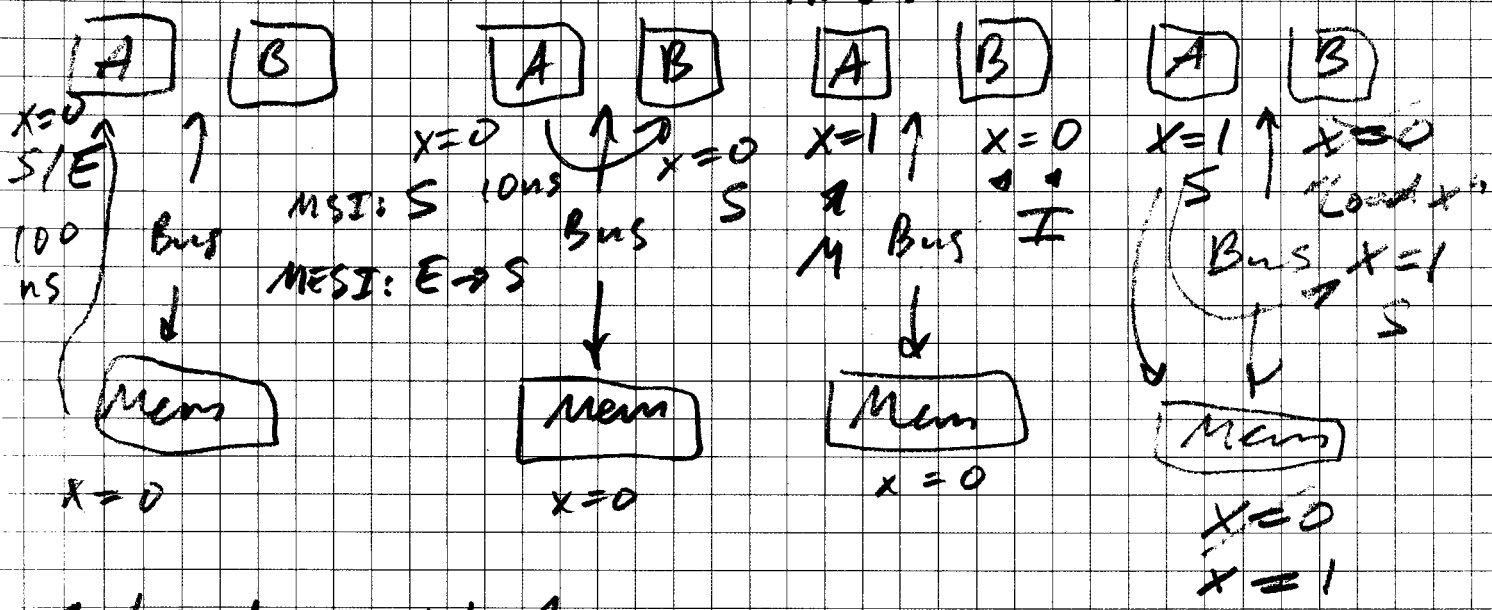
At time step 4, what happens if CPU B tries to read X?

- if CPU A reads X it will see 1
- CPU B can no longer see X as 0
- A write of X=2 by CPU B means a CPU cannot see X as 2 and later as 1

Solving the Initial Problem:

ex. coherent solution to prev problem

Time 0, everything I



* A is changing x to 1
* B now an invalidated value

Cache Coherence: what values can be returned on a read

Cache Coherence Protocols:

we will focus on a snooping, write-invalidate protocol with centralized, shared memory

Snooping: caches track state of their cache blocks, "snoop" on shared bus for updates

Write-invalidate: a processor must have sole access of a block before writing it

Protocol 1 : MSI

Modified : block is a private copy, has been modified

Shared : block is potentially a shared copy, unmodified (wrt memory)

Invalid : valid bit = 0 for this cache block

Protocol 2 : MESI 4 states for this class of
, Modified, Exclusive, Shared, Invalid

Exclusive : block is private, unmodified

Note. There exist MOESI, MESIF. IBM uses
13 states!

Complete Ex :

MESI, write-invalidate protocol, centralized/
shared memory, write-back $

Two processors : P0, P1

P0 : cache

Index	Coherence State	Tag	Word 1	Word 0
0	I/E/S/M/S	4	—	—
1	I			
2	I			
3	I			

P1's cache

Index	Coherence State	Tag	Word 1	Word 0
0	E/S/I/S	4	—	—
1	I			
2	I			
3	E/M/E	28		

↳ modified wrt memory

Event	Effects
P0 reads from addr with index=0, tag=4 (read miss)	P0 places IOT4 read request on bus • P1 ignores • Mem sends IOT4 to P0
P1 reads from addr w/ index 0, tag 4 (read miss)	P1 places IOT4 read req. on bus • P0 sends to P1 (P0, P1 in S) • cancel mem transfer, S
P0 writes to addr w/ index 0, tag 4 (write hit)	P0 broadcasts a write invalidate (IOT4) • P1 → I • P0 → M
P0 reads from addr w/ index 0, tag 4 (read hit)	P0 does not broadcast anything
P1 writes to addr w/ index 3 tag 2 (write miss)	P1 places I3T2 write request on the bus • P0 ignores • Mem sends I3T2 to P1
P1 reads from addr w/ index 0 tag 4 (read miss)	P1 places IOT4 read request on bus • P0 sends to P1 (both in S) • P0 also sends to memory
P1 reads from addr w/ index 3, tag 8 (read miss)	P1 places I3T8 read req to bus • P0 ignores • Memory sends to P1 • I3T2 goes to memory

General Guidelines:

Read miss, write miss: → Broadcast (is invalid)

Hit → read → directly reads to known no broadcast

→ write → shared → broadcast

→ modified exclusive → Private, no broadcast

Final Exam Review

12.9.25
L28

* All Exam Topics in Slides *

Cache Coherence Complete Ex.

MESI, write-invalidate protocol, centralized/shared memory, write-back $, two processors P0, P1

P0's cache:

Index	Coherence State	Tag	Word 1	Word 0
0	S	2	—	—
1	I			
2	E	9	—	—
3	I			

P1's cache:

Index	Coherence State	Tag	Word 1	Word 0
0	E	8	—	—
1	I			
2	~~S~~ M	2	—	—
3	~~I~~ E	9	—	—

Event	Effects
P0 reads from addr w/ I0T2 • read hit	None, no bus traffic
P1 writes to addr w/ I2T2 • write hit	Broadcast invalidate P0 takes no action P1, I2T2 → M
P1 reads from addr with I3T9 • read miss	Request I3T9 on bus Memory sends on bus P1, I3T9 → E

Hazards:

Solution: register renaming

WAW: add x10, x11, x12 → add T4, ..., ...
sub x10, x15, x10 → sub TN, ..., ...
(xor x12, x10, x19) (xor ..., TN, ...)

WAR: and x3, x7, x15 → and ..., ..., tx
or x15, x10, x22 → or ty, ..., ...

RAW: lw x17, 0(x18) → addi x4, x17, x9
] cannot reorder, data flow exists b/w 2 insts

RAR: NOT a hazard

note. load-use hazard is a RAW hazard* control hazards still exist!

note. sub x10, x11, x12 → add x15, x10, x8
] RAW hazard if no forwarding

* EVEN with forwarding!