

CMPE 200
Computer Architecture & Design

Midterm Review (1)

Haonan Wang



SAN JOSÉ STATE
UNIVERSITY

Midterm Exam Composition

- **Multiple-choice Single-answer: 4 points * 8 = 32**
- **Multiple-choice Multiple-answer: 6 points (wrong or missing option -3) * 4 = 24**
- **True or False: 3 points * 8 = 24**
- **Questions & Answers: 10 points (steps 8 + answer 2) * 2 = 20**
- **Bonus question: 10 points * 1 = 10**

Total = 32 + 24 + 24 + 20 + 10 = 100 + 10

- If you get more than 100 points, the excessive points will be used to improved your overall grade.

Exam settings:

- Time: Monday, Oct 19, 15:00 - 16:15 @ Clark 222 with LockDown Browser
- Closed book (but you can print out the MIPS data card and use it if needed)
- Cheat sheet (handwritten): A4 paper * 1 allowed, pictures of both sides must be submitted before the exam
- Use of calculator and scratch paper allowed
- Double-check your laptop and turn off your cellphone before the exam

Relative Performance

- Define Performance = 1/Execution Time
- “X is n times faster than Y”

$$\begin{aligned} & \text{Performance}_X / \text{Performance}_Y \\ &= \text{Execution time}_Y / \text{Execution time}_X = n \end{aligned}$$

Example: time taken to run a program -- 10s on a computer A, 15s on computer B

$$\text{Execution Time}_B / \text{Execution Time}_A = 15\text{s} / 10\text{s} = 1.5$$

A is 1.5 times faster than B.

Example: Average CPI

Instruction Type	Integer	Floating point	Branch	Load/Store
CPI for type	1	2	4	3
Instruction Count in program A	50	10	10	10
Instruction Count in program B	20	0	10	10

- Program A: Total Instructions = 80

$$\text{Clock Cycles} = 50 \times 1 + 10 \times 2 + 10 \times 4 + 10 \times 3 = 140$$

$$\text{Avg. CPI} = 140/80 = 1.75$$

$$\begin{aligned} \text{CPU time (if Clock period} &= 100 \text{ ps)} \\ &= 1.75 \times 80 \times 100 \text{ ps} = 14000 \text{ ps} = 14 \text{ ns} \end{aligned}$$

- Program B: Total Instructions = 40

$$\text{Clock Cycles} = 20 \times 1 + 10 \times 4 + 10 \times 3 = 90$$

$$\text{Avg. CPI} = 90/40 = 2.25$$

$$\begin{aligned} \text{CPU time (if Clock period} &= 100 \text{ ps)} \\ &= 2.25 \times 40 \times 100 \text{ ps} = 9000 \text{ ps} = 9 \text{ ns} \end{aligned}$$

Throughput

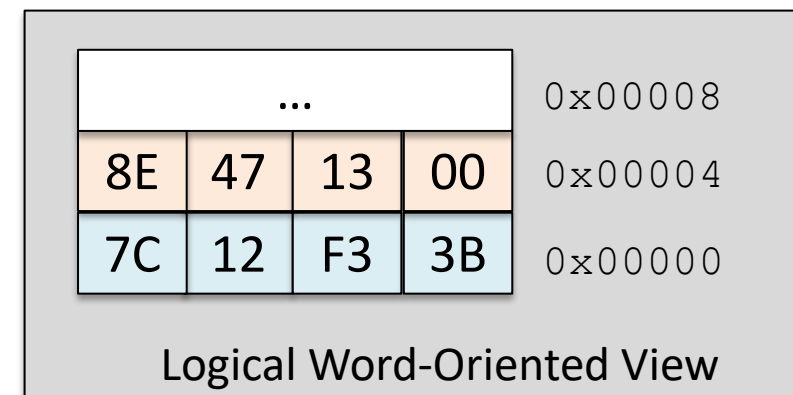
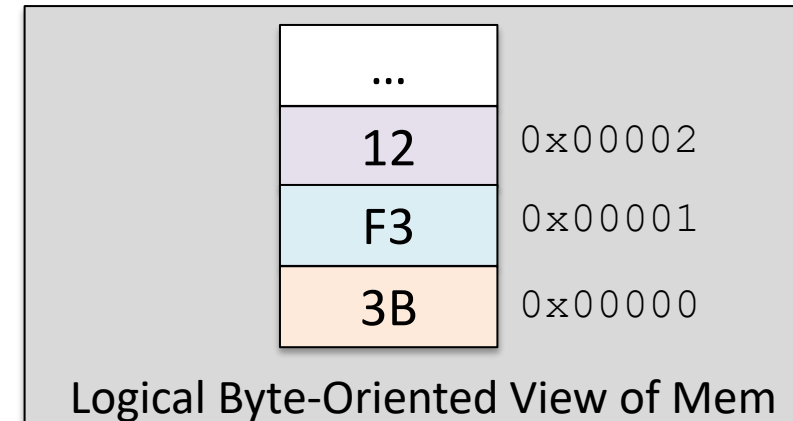
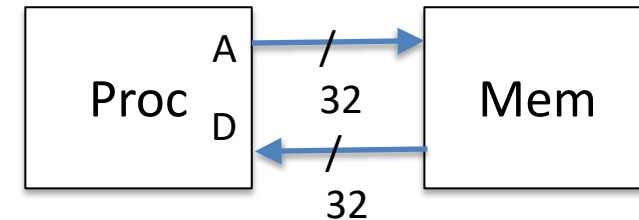
- **Throughput (T) = # of instructions executed / time**
- **IPC = Instruction Per Cycle = 1 / CPI**
 - For a large number of instructions, the IPC of a pipelined processor is **1 instruction per clock cycle**
 - **Only when we keep the pipeline full of instructions**

Assume we execute **N instructions** using a **K stage** datapath

Pipeline IPC	$N / (N+K-1)$
Let $n \rightarrow \infty$ ($n = \text{infinity}$)	1

Byte-oriented vs. Word-oriented Memory

- **Most processors are byte-oriented**
 - Can access a word from any byte address
- **MIPS: Word-oriented**
 - Words must be aligned to multiples of 4
 - **Not word-addressable!**
 - Provides some simplicity in design
- Logical views can be arranged in **rows of 4-bytes** for word-oriented memories



Tips to Remember Endianness

SIMPLY EXPLAINED



0xCAFEBAFE
will be stored as
CA | FE | BA | BE



Looks normal (same as
writing order) when address
is from low to high

0xCAFEBAFE
will be stored as
BE | BA | FE | CA



Looks strange here
But same as writing order when
address is from high to low

word value:

0 x 1 2 3 4 5 6 7 8

0x03	12	0x00	78
0x02	34	0x01	56
0x01	56	0x02	34
0x00	78	0x03	12

Little-Endian

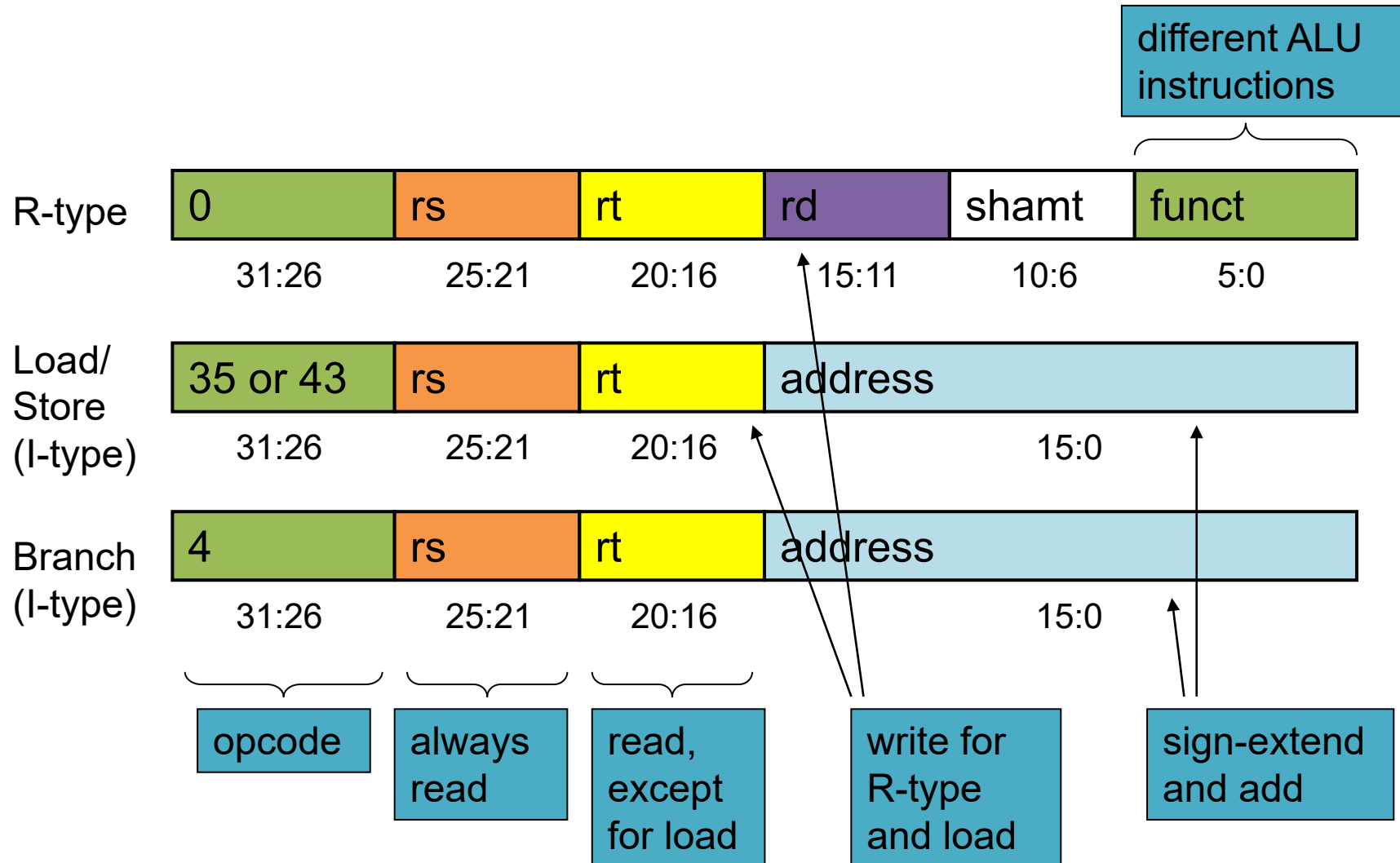
...	0x00004
12 34 56 78	0x00000

Logical Word-Oriented View

Signed vs. Unsigned Instructions

- **Other instructions also have unsigned versions**
 - add_u, addi_u, sub_u, div_u, mult_u, ...
- **Example:**
 - Assume that \$2 = 0xFFFFFFFF, \$3 = 0x00000001, what is \$1?
 - slt \$1, \$2, \$3 # signed set less than
 # -1 < +1 → \$1 = 1
 - sltu \$1, \$2, \$3 # unsigned set less than
 # +4,294,967,295 > +1 → \$1 = 0

Instruction Formats



Branch Target Addressing

- **Why branch's next instruction should be considered?**
 - Because MIPS increments program counter (PC) by 4 before executing an instruction
 - This already incremented PC value is used for branch target address calculation
- **Branch Target Address Calculation**
 - **Target address = branch's next instruction address + offset x 4**
 - **bne \$t0, \$s5, Exit**
 - Exit (0x00080018) = addi address (0x00080010) + distance (2) x 4 byte/inst
 - **b Loop**
 - Loop (0x00080000) = Exit address (0x00080018) + distance (-6) x 4 byte/inst

80000	Loop: sll \$t1, \$s3, 2
80004	add \$t1, \$t1, \$s6
80008	lw \$t0, 0(\$t1)
8000C	bne \$t0, \$s5, Exit
80010	addi \$s3, \$s3, 1
80014	b Loop
80018	Exit: ...

Loading an Immediate

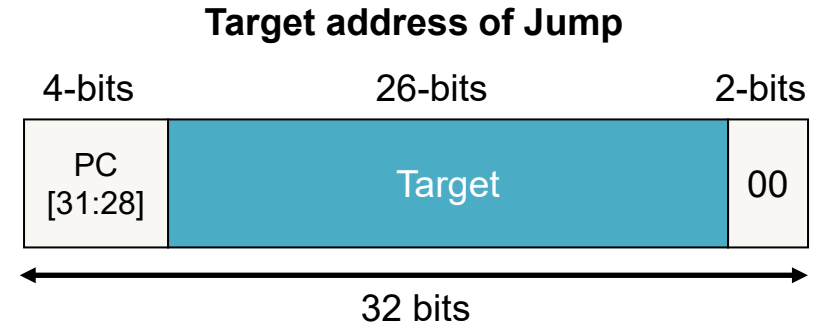
- What if you want to load an immediate value to a register?
- If immediate (constant) is 16 bits or less
 - Use **ori** or **addi** instruction with \$0 register
 - Examples : You want to load value 1 to \$2
 - `addi $2, $0, 1` // $R[2] = 0 + 1 = 1$
 - `ori $2, $0, 0x1` // $R[2] = 0 \mid 1 = 1$
- If immediate is more than 16 bits
 - Immediates limited to 16 bits so we must load constant with a 2-instruction sequence using the special **LUI (Load Upper Immediate)** instruction
 - To load \$2 with 0x12345678
 - `lui $2, 0x1234`
 - `ori $2, $2, 0x5678`

LUI: the immediate value is loaded to the MSB 16 bits of the target register

R[2]	12340000	LUI
	OR 00005678	
R[2]	12345678	ORI

Jump Target Addressing

- Jump instruction provides larger scale jump than branch
- Target address = First 4 bits of jump's next instruction address : Last 28 bits of (target x 4)
- Example: j Loop (0x00080000)
 - The first 4 bits of Exit = 0x0
 - Last 28 bits of target x 4 = 0080000
 - target = 0x0020000



```
00080000  Loop: sll $t1, $s3, 2
00080004      add $t1, $t1, $s6
00080008      lw  $t0, 0($t1)
0008000C      bne $t0, $s5, Exit
00080010      addi $s3, $s3, 1
00080014      j   Loop
00080018  Exit: ...
```

Exercise

- Translate the given high-level language code to MIPS assembly.
 - Assume that the address of integer variable x and y are in \$4 and \$1 respectively.

High-level language

```
if (x > y)
    x = x + y;
else
    x = 1;
```

Tasks to do:

- 1) Load x value to a register
- 2) Load y value to another register
- 3) Check if y is less than x
- 4) If true, run add operation
- 5) Else, store 1 to one register
- 6) Store the result value to a in memory

MIPS

```
lw      $2, 0($4)
lw      $3, 0($1)
slt     $1, $3, $2
beq     $1, $0, Else
add     $2, $2, $3
b       Next
addi    $2, $0, 1
sw      $2, 0($4)
```

Exercise 2

- **Translate the given high-level language code to MIPS assembly**
 - Assume that the base address of a **char** array a is in register \$s0
 - base address of an array: the address of the first element of the array
 - Assume that the value of i is in \$t0.

High-level language

a[i]--;

MIPS

```
add $t2, $s0, $t0
lb  $t3, 0($t2)
subi $t3, $t3, 1
sb  $t3, 0($t2)
```

Now we know that the address of a[i] is
 $\$s0 + 1\text{byte} * i = \$s0 + \$t0$

Is this a valid operation to get one byte
from $\$s0 + \$t0$?

lb \$t1, \$t0(\$s0)

No, because offset should be an
immediate value, not a register id

→ We should change the base address
value, not offset value
i.e. new base: $\$t2 = \$s0 + \$t0$
and then load one byte from 0(\$t2)

Less Than Comparisons

- To reduce the number of instructions, you can also use `slti` or `sltui` instead of `slt`

Example: Add numbers from 0 to 9

High-level language

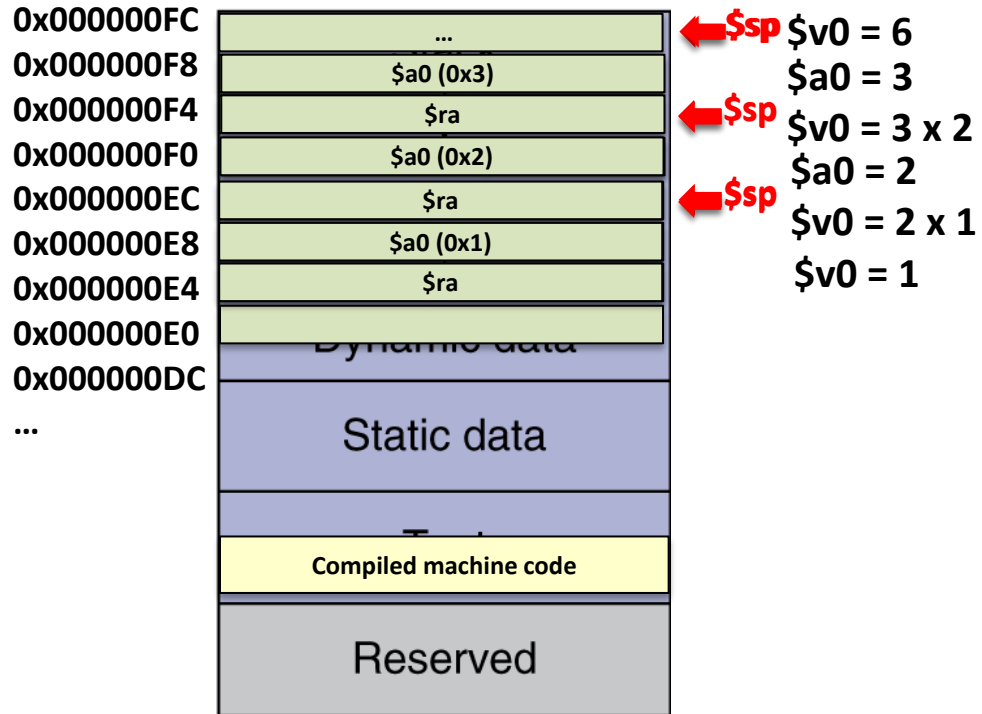
```
int sum = 0;
int i;
for (i = 0; i < 10; i++)
{
    sum = sum + i;
}
```

MIPS

```
# $s0 = i, $s1 = sum

add    $s0, $0, $0
add    $s1, $0, $0
for:   slti    $t1, $s0, 10
       beq    $t1, $0, done
       add    $s1, $s1, $s0
       addi   $s0, $s0, 1
       j      for
done:
```

Example: Recursion for 3!

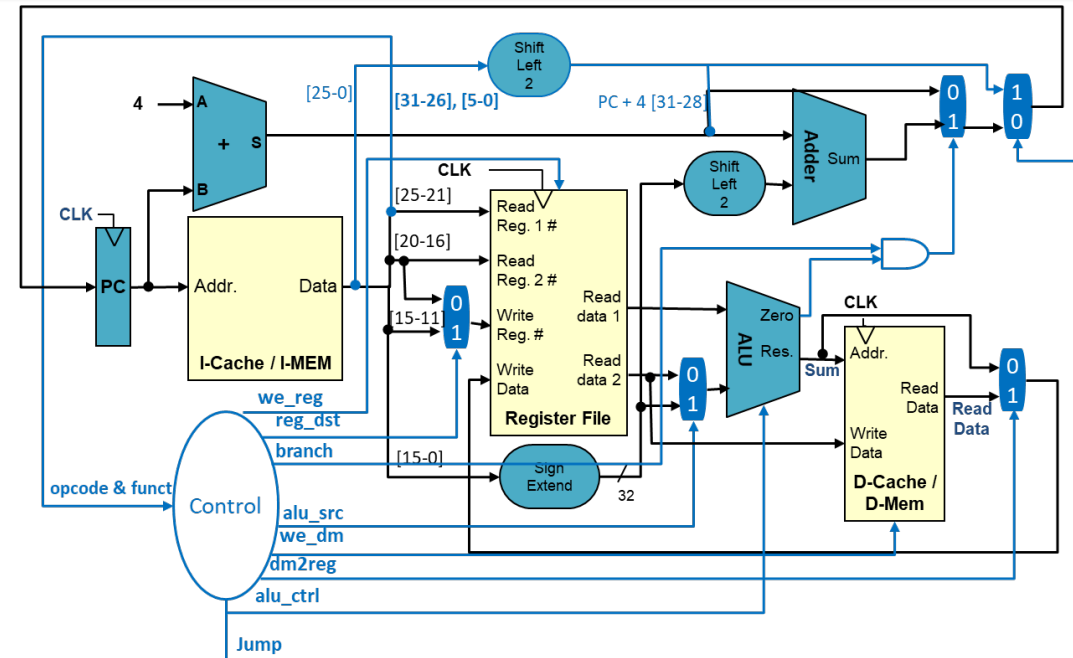


```
factorial:  addi  $sp, $sp, -8  # make room
            sw    $a0, 4($sp) # store input (n)
            sw    $ra, 0($sp) # store return address
            addi  $t0, $0, 2   # $t0 = 2
            slt   $t0, $a0, $t0 # n <= 1?
            beq   $t0, $0, else # no: go to else (recursion)
            addi  $v0, $0, 1   # yes: return 1
            addi  $sp, $sp, 8   # restore $sp
            jr    $ra          # return

else:       addi  $a0, $a0, -1  # n = n - 1
            jal   factorial    # recursive call
            lw    $ra, 0($sp)  # restore return address
            lw    $a0, 4($sp)  # restore input
            addi  $sp, $sp, 8   # restore $sp
            mul   $v0, $a0, $v0 # n * factorial(n-1)
            jr    $ra          # return
```


Single-Cycle CPU Performance Analysis

Function Unit	Parameter	Delay (ps)
Register clock-to-Q	T_{pcq_PC}	30
MUX	T_{MUX}	25
Sign-extend	T_{s_ext}	25
ALU	T_{ALU}	200
Mem read	T_{mem}	250
Register file read	T_{RRead}	150
Register file write	$T_{RFwrite}$	20

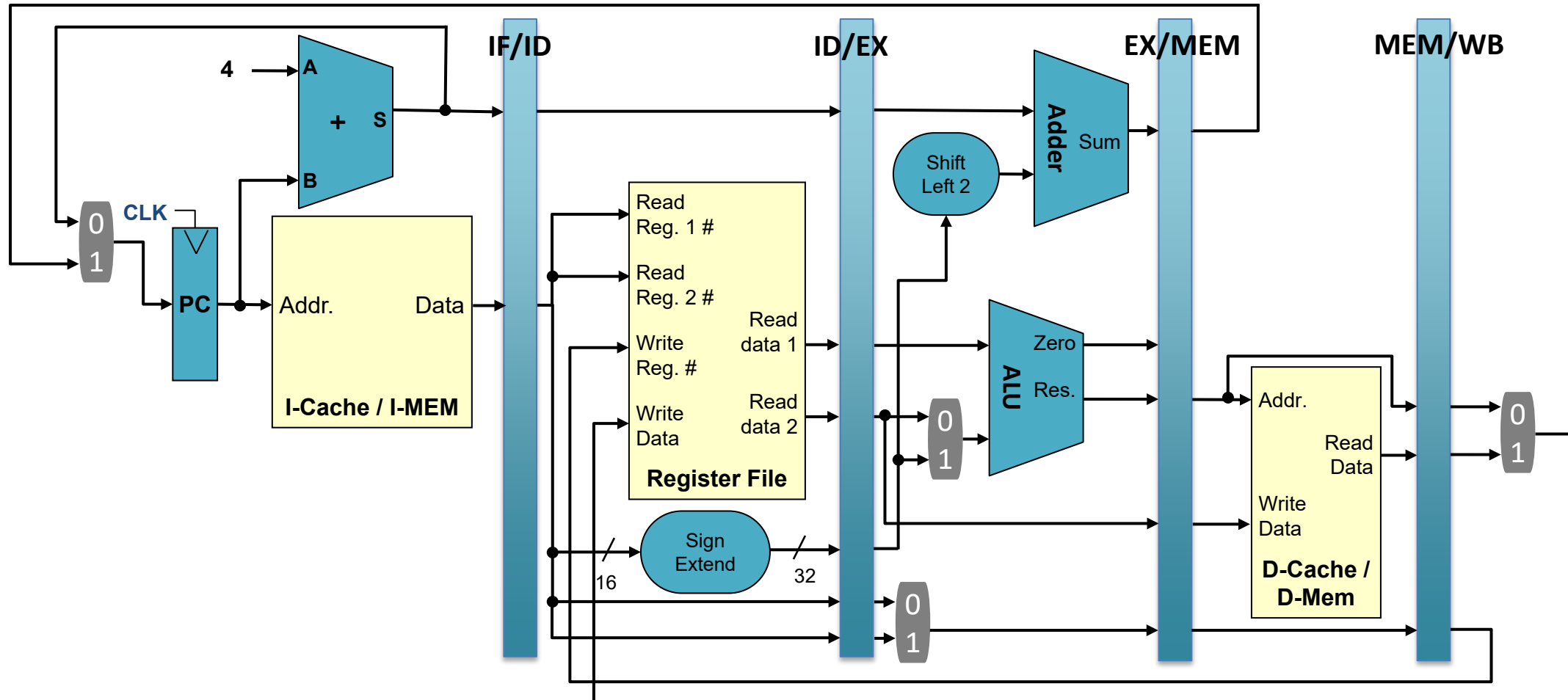


Critical path for LW =

$$= (30 + 250 + 150 + 200 + 250 + 25 + 20) \text{ ps}$$

$$= 925 \text{ ps}$$

5-Stage Pipelined CPU



Pipelined Timing

- Suppose we execute **N instructions** using a **K stage** datapath
- Total execution cycle: **$K+N-1$ cycles**
 - K cycle for 1st inst + (N-1) cycles for remaining insts
 - Assume we keep the pipeline full

**7 Instrs. =
11 clocks (5 + 7 – 1)**

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10	CC11
LW	IF	ID	EXE	MEM	WB						
ADD		IF	ID	EXE	MEM	WB					
SUB			IF	ID	EXE	MEM	WB				
SW				IF	ID	EXE	MEM	WB			
AND					IF	ID	EXE	MEM	WB		
OR						IF	ID	EXE	MEM	WB	
XOR							IF	ID	EXE	MEM	WB

Data Hazards

- **Read-After-Write (RAW)**
 - A following instruction reads a result from a previous instruction
- **Write-After-Read (WAR)**
- **Write-After-Write (WAW)**
- **RAW Example**
 - LW **\$t1**,4(\$s0)
 - ADD \$t5,**\$t1**,\$t4

Initial Conditions:

\$s0 = 0x10010000

\$t1 = 0x0

\$t4 = 0x24

\$t5 = 0x0

00000060	0x10010004
12345678	0x10010000

After execution values should be:

\$s0 = 0x10010000

\$t1 = 0x60

\$t4 = 0x24

\$t5 = 0x84

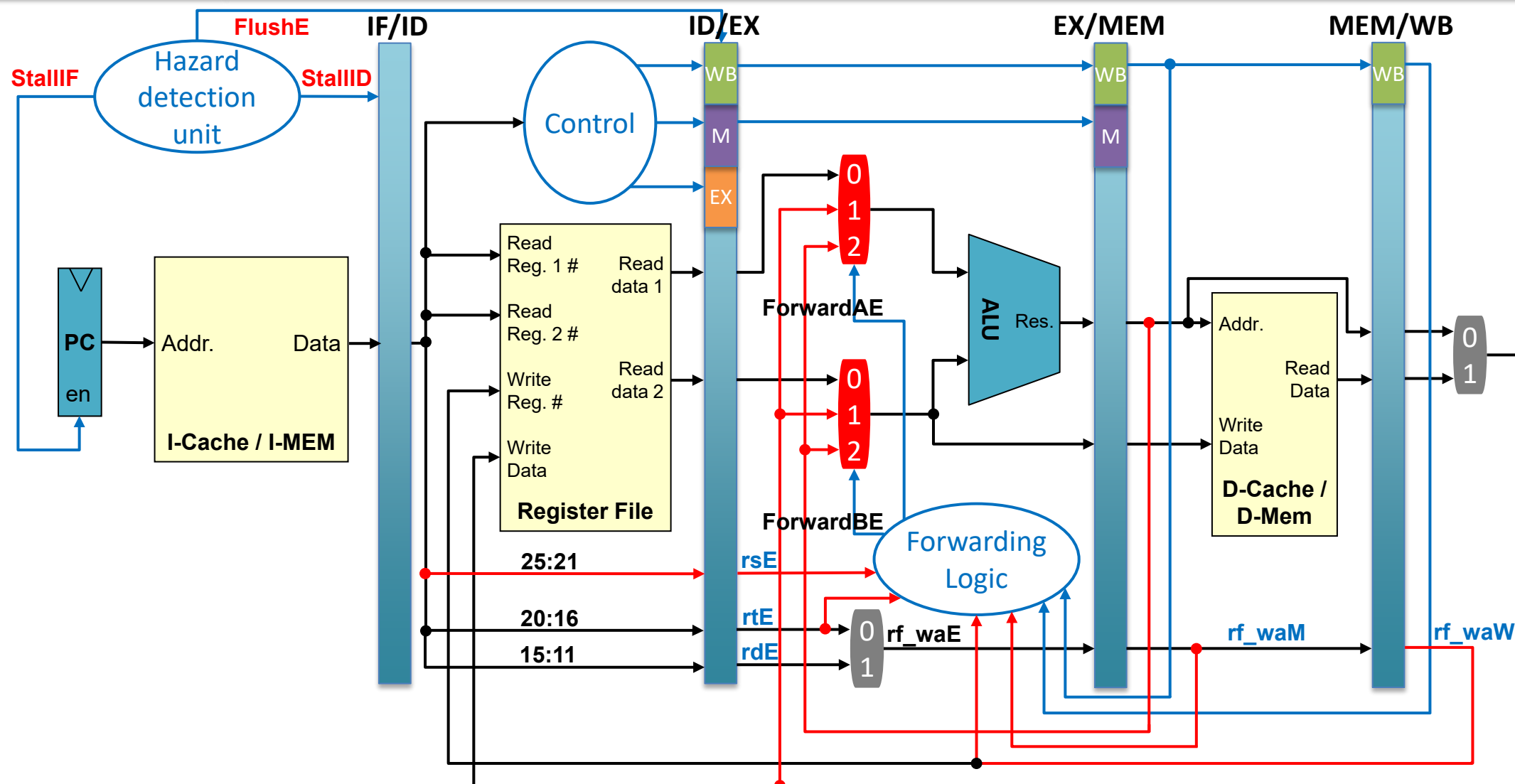
Pipeline Latency with Forwarding/Stall

- **Number of cycles we must stall to execute a dependent instruction:**

Instruction that is the dependency source	w/o Forwarding (cycles)	w/ Forwarding (cycles)
LW	2	1
Other instructions that update register file	2	0

- **Data Hazards Solutions:**
 - Hardware: stalling & forwarding
 - Software: compiler -- code reordering

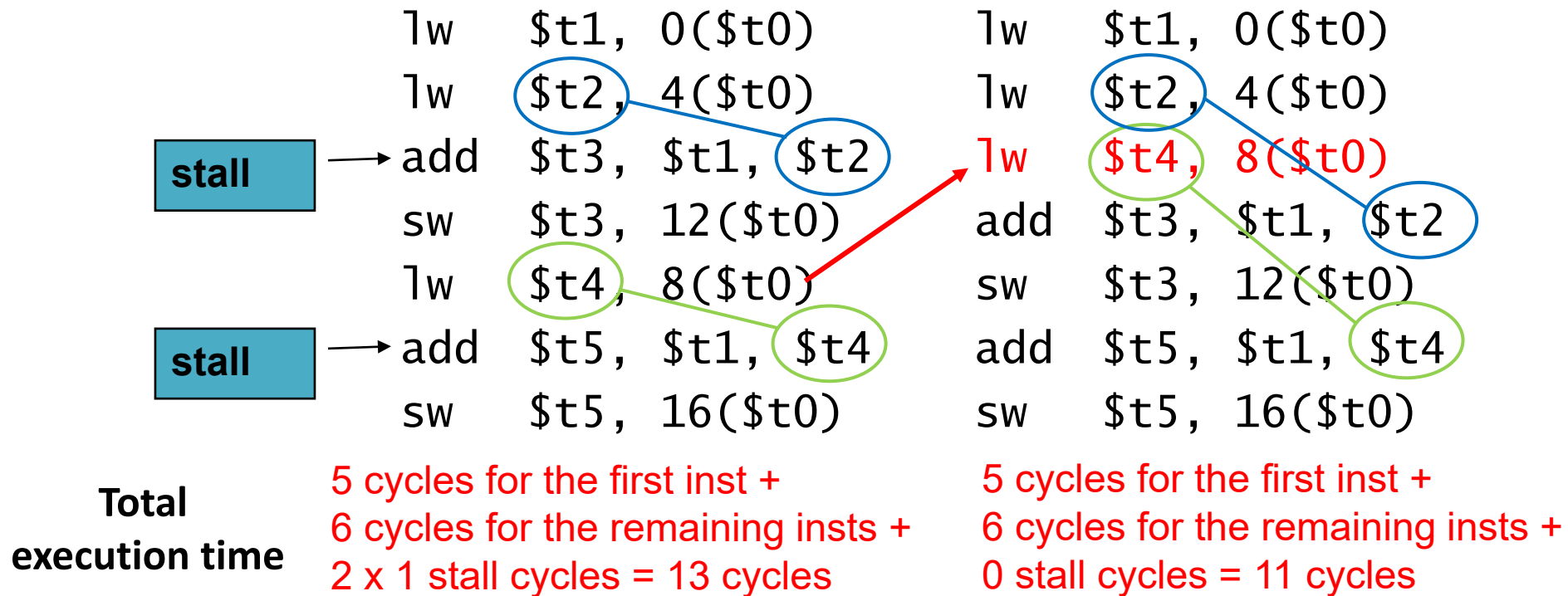
Forwarding Path



Note: Logics for sign extensions and next pc calculation are omitted for simplicity

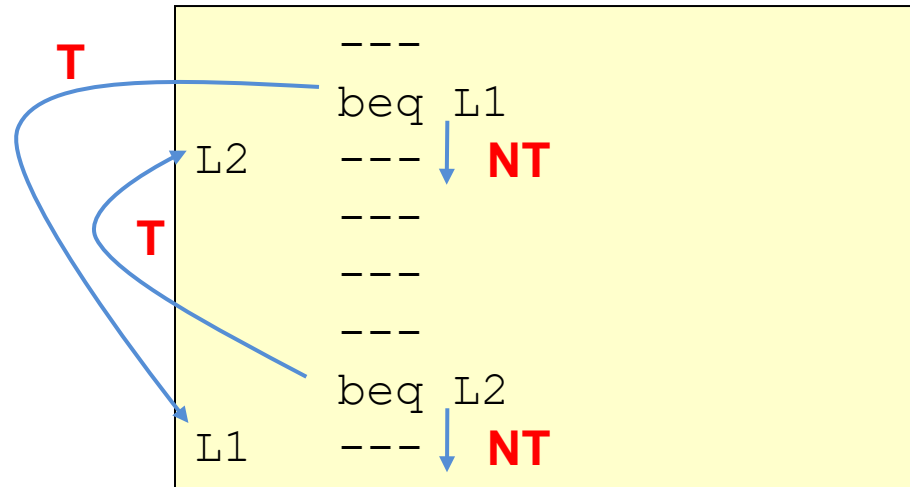
Code Scheduling to Avoid Stalls

- Compiler can reorder code to avoid the stalls
- C code for $A = B + E$; $C = B + F$;



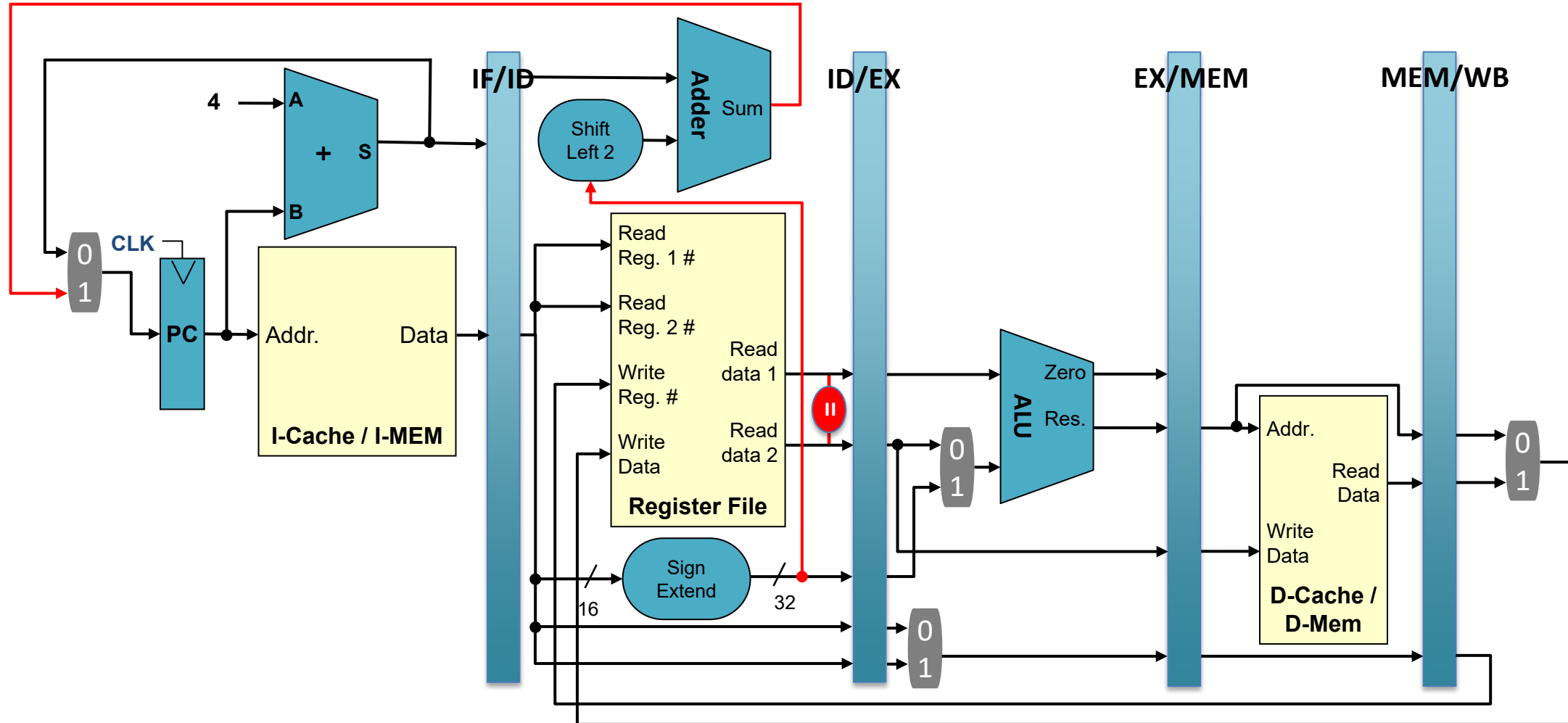
Control Hazard

- **Branch outcomes: Taken (T) or Not-Taken (NT)**
- **Not known until late in the pipeline**
 - Prevents us from fetching future instructions
 - Rather than stall, we can predict the outcome and keep fetching
 - We only need to correct the pipeline if we guess wrong
- **Static Predictions**
 - Predict Taken
 - Predict Not Taken



Early Branch Determination

- Target address can be calculated earlier by moving shift-left-2 and Adder



Branch Delay Slot

Actual Branch Outcome

```
BEQ $a0,$a1,L1 (NT)
L2: ADD $s1,$t1,$t2
SUB $t3,$t0,$s0
BNE $a0,$s1,L2 (T)
OR  $s0,$t6,$t7
L1: AND $t3,$t6,$t7
SW  $t5,0($s1)
LW  $s2,0($s5)
```

Instruction in the target address (L2)

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10
BEQ	IF	ID	EXE	MEM	WB					
ADD		IF	ID	EXE	MEM					
SUB			IF	ID	EXE	MEM	WB			
BNE				IF	ID	EXE	MEM	WB		
OR					IF	ID	EXE	MEM	WB	
ADD						IF	ID	EXE	MEM	WB
SUB							IF	ID	EXE	MEM





No Flush & Better Performance

Early Determination w/ Predict NT

Actual Branch Outcome

BEQ \$a0,\$a1,L1 (NT)
 L2: ADD \$s1,\$t1,\$t2
 SUB \$t3,\$t0,\$s0
 OR \$s0,\$t6,\$t7
 BNE \$a0,\$s1,L2 (T)
 L1: AND \$t3,\$t6,\$t7
 SW \$t5,0(\$s1)
 LW \$s2,0(\$s5)

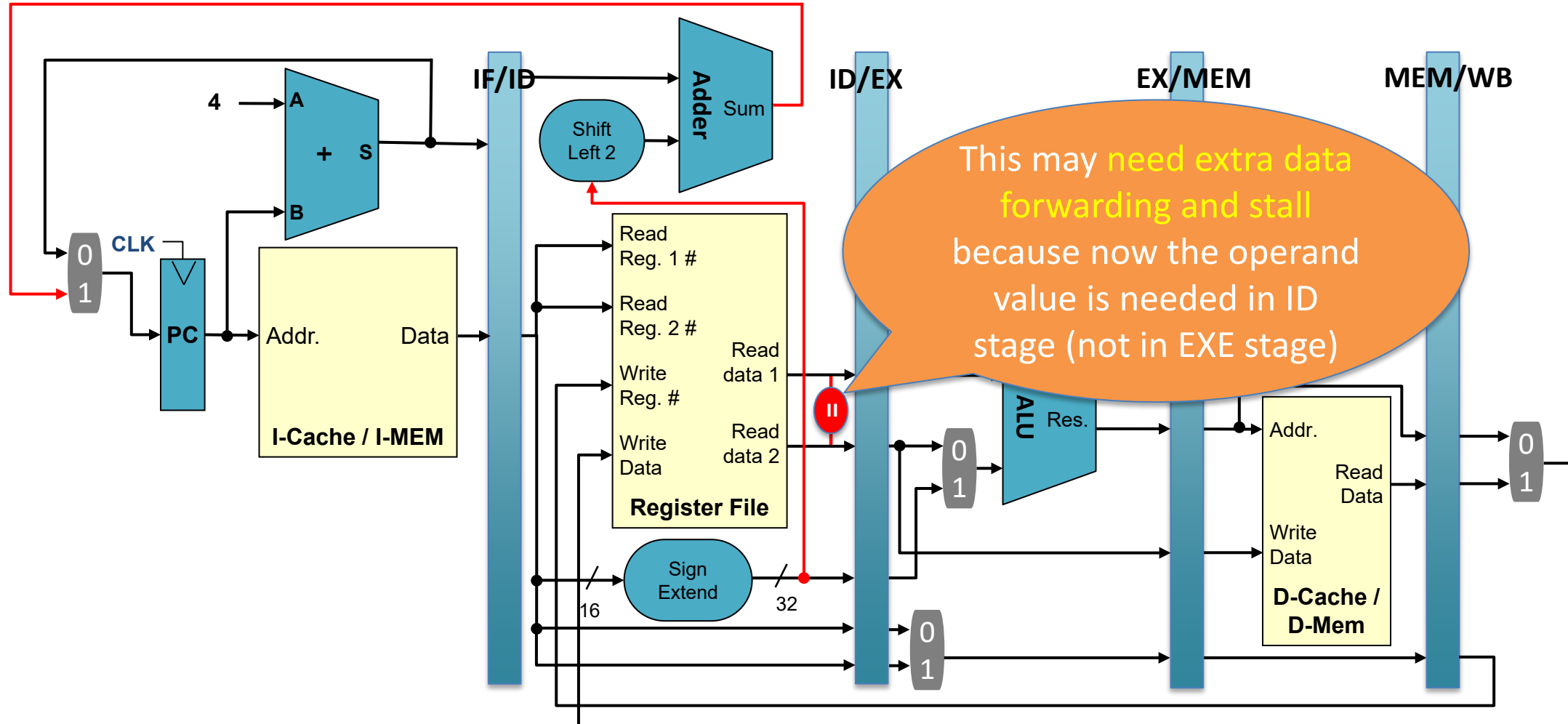
Instruction in the
target address (L2)

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9	CC10
BEQ	IF	ID	EXE	MEM	WB					
ADD		IF	ID	EXE	MEM	WB				
SUB			IF	ID	EXE	MEM	WB			
OR				IF	ID	EXE	MEM	WB		
BNE					IF	ID	EXE	MEM	WB	
AND						IF				
ADD							IF	ID	EXE	MEM
SUB								IF	ID	EXE

Flush

Early Branch Determination: Issues

- Target address can be calculated earlier by moving shift-left-2 and Adder



Early Determination w/ Predict NT

Actual Branch Outcome

BEQ \$a0,\$a1,L1 (NT)
 L2: ADD \$s1,\$t1,\$t2
 SUB \$t3,\$t0,\$s0
 OR \$s0,\$t6,\$t7
 BNE **\$s0**,\$s1,L2 (T)
 L1: AND \$t3,\$t6,\$t7
 SW \$t5,0(\$s1)
 LW \$s2,0(\$s5)

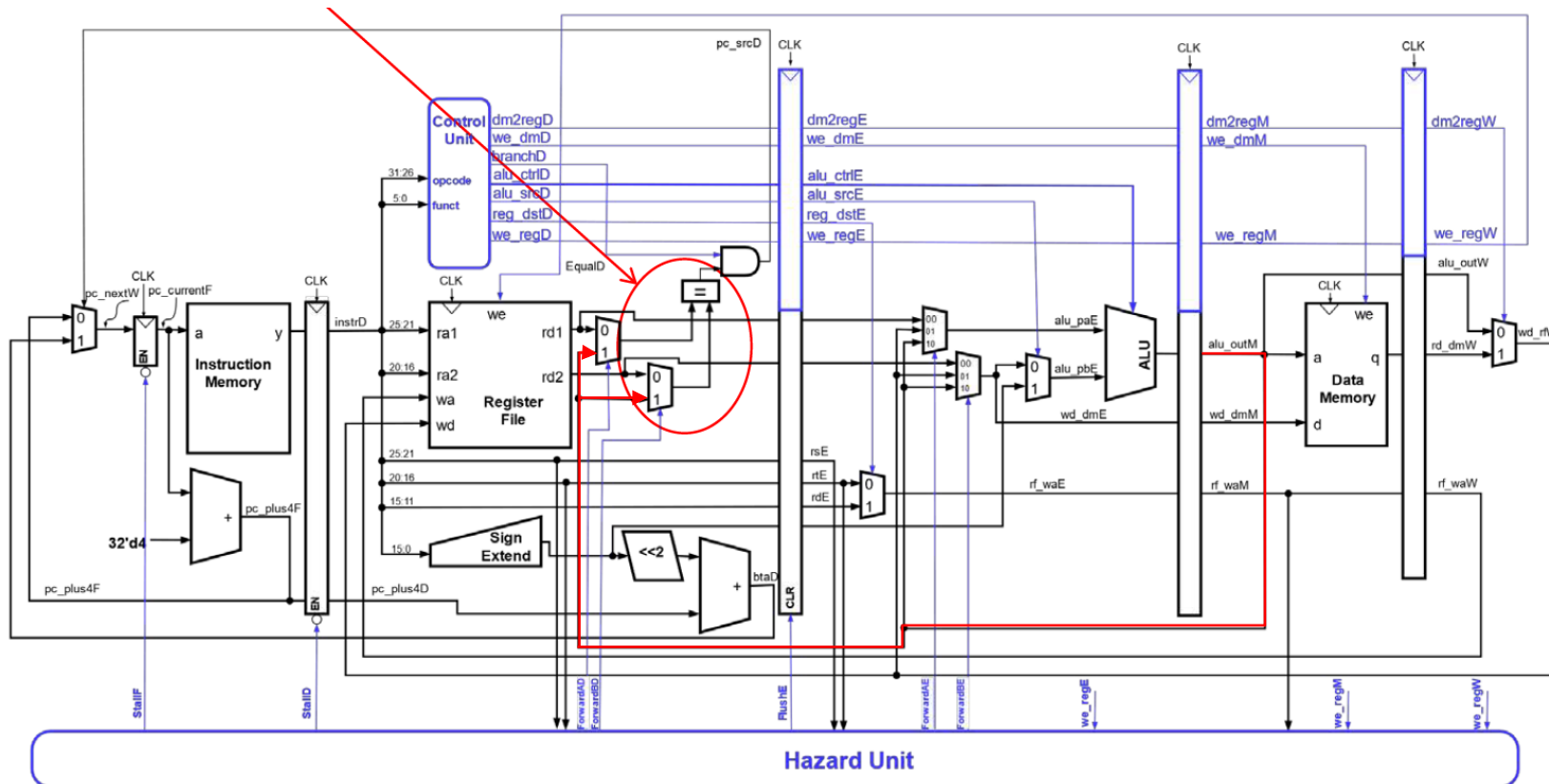
Instruction in the
target address (L2)

	CC1	CC2	CC3							
BEQ	IF	ID	EXE							
ADD		IF	ID							
SUB			IF	ID	EXE					
OR				IF	ID	EXE			WB	
BNE					IF	ID	ID	EXE	MEM	WB
AND						IF	IF	nop	nop	nop
ADD								IF	ID	EXE
SUB									IF	ID

If BNE has dependency with its preceding instruction, OR, one cycle stall is required to get the OR's result value
 → Still better than normal 3 cycle flushing

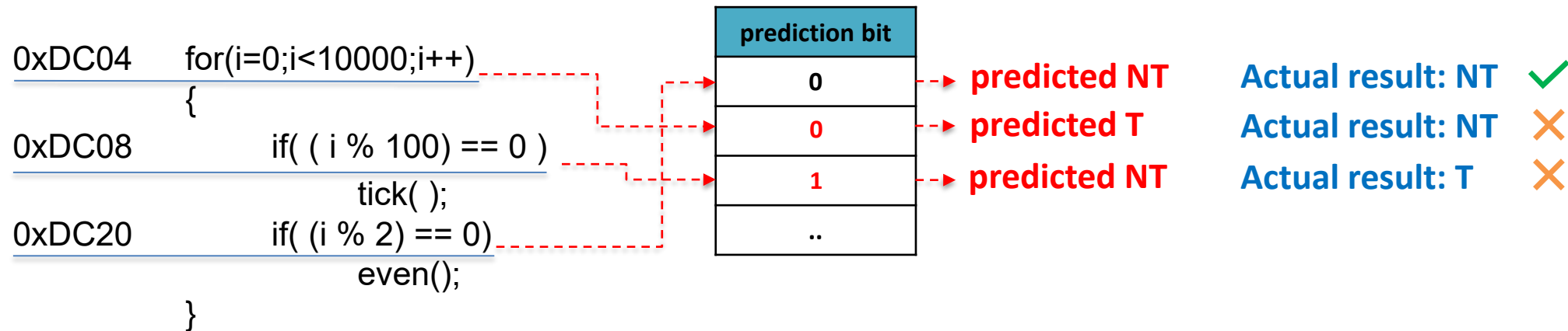
Early Branch Determination

- Forwarding logic for early branch determination
 - $\text{ForwardAD} = \text{branchD}$ and we_regM and $(\text{rsD} \neq 0)$ and $(\text{rf_waM} == \text{rsD})$
 - $\text{ForwardBD} = \text{branchD}$ and we_regM and $(\text{rtD} \neq 0)$ and $(\text{rf_waM} == \text{rtD})$



1-bit Predictor

- **Each entry of branch prediction buffer is 1-bit (1: Taken, 0: Not Taken)**
 - The entry value is the latest outcome of the branch
 - Next outcome of the branch is predicted based on current value
 - Example: Assume that the actual outcomes of the branches at **0xDC04**, **0xDC08**, and **0xDC20** are **untaken**, **taken**, and **untaken**, respectively



Is 1 bit Enough?

```
0xDC04    for(i=0;i<10000;i++)
           {
0xDC08           if( ( i % 100) == 0 )
                    tick( );
0xDC20           if( (i % 2) == 0)
                    even();
           }
```

Not taken and continue to the loop body

Taken to exit the loop

NT

TN

DC04:

Diagram illustrating a sequence of iterations. A horizontal line represents the sequence, with "NNNNNNNN" repeated on both sides of a central point, separated by "...". A red double-headed arrow below the line spans the first "NNNNNNNN" and is labeled "10,000 iterations". A blue double-headed arrow below the line spans the second "NNNNNNNN" and is labeled "10,000 iterations ...".

Mis-predictions: 2 / 10,000

Mis-predictions for every first and last iterations
→ **99.998% Correct Prediction**

DC08:

TTTTT ... TNTTTT ... TNTTTT ...

Mis-predictions: 2 / 100

98.0% Correct Prediction

DC20:

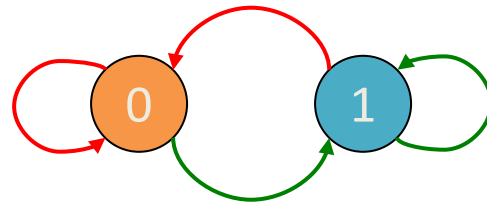
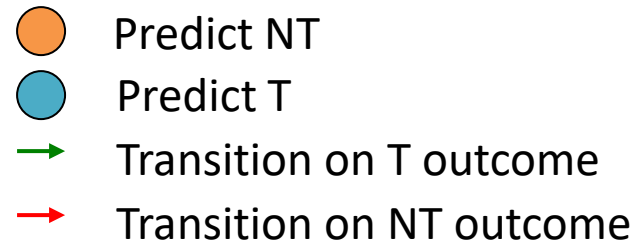
[illegible]

Mis-predictions: 2 / 2

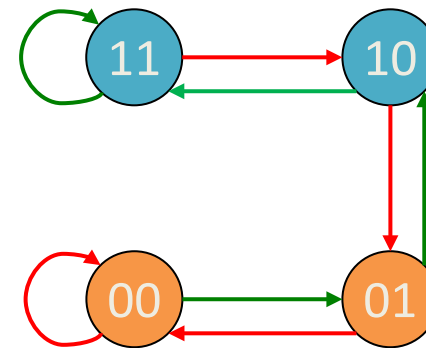
0.0% Correct Prediction

Using 2-bit History

- **2-bit Saturating Counter in each branch prediction buffer entry**
 - Could have more than two bits but two bits cover most patterns (i.e. loops)



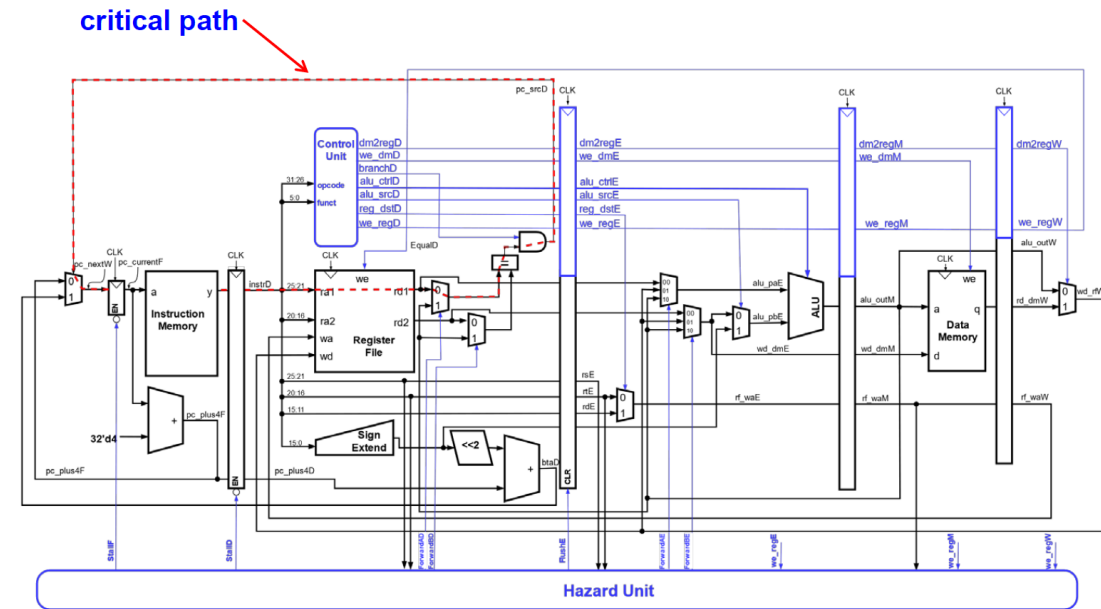
FSM for 1-bit
Prediction



FSM for 2-bit
Saturating Counter

Pipelined CPU Performance Analysis

Function Unit	Parameter	Delay (ps)
Register clock-to-Q	T_{pcq}	30
Clock setup time	T_{setup}	20
MUX	T_{MUX}	25
Sign-extend	T_{s_ext}	25
ALU	T_{ALU}	200
Mem read	T_{mem}	250
AND	T_{AND}	15
EQ	T_{eq}	40
Register file read	T_{Rfread}	150
Register file write	$T_{RFwrite}$	20



Critical path = max {

$$\begin{aligned}
 &T_{pcq} + T_{i-mem} + T_{setup}, \\
 &2(T_{Rfread} + T_{mux} + T_{eq} + T_{AND} + T_{mux} + T_{setup}), \\
 &T_{pcq} + T_{mux} + T_{mux} + T_{ALU} + T_{setup}, \\
 &T_{pcq} + T_{memread} + T_{setup}, \\
 &2(T_{pcq} + T_{mux} + T_{RFwrite})
 \end{aligned}$$

}

RF write and read need to be finished in the first and second half of each cycle, respectively

Question: Pipelined CPI

- **SPECINT2000 benchmark:**

- 25% loads, 10% stores, 11% branches, 2% jumps, 52% R-type

- **Assume:**

- 40% of loads used by next instruction
- 25% of branches mispredicted
- All jumps flush next instruction

- **Average CPI calculation:**

- Load/Branch CPI = 1 when no stalling, 2 when stalling
- $CPI_{lw} = 1(0.6) + 2(0.4) = 1.4$
- $CPI_{beq} = 1(0.75) + 2(0.25) = 1.25$
- **Overall average CPI = $(0.25 \times 1.4) + (0.1 \times 1) + (0.11 \times 1.25) + (0.02 \times 2) + (0.52 \times 1) = 1.15$**

Average CPI

= Total Clock Cycles / Total IC

= $\sum IC_i \times CPI_i$ / Total IC

= $\sum IR_i \times CPI_i$

CPU Performance Comparison

- For a program with 100 billion instructions executing on the pipelined MIPS processor under discussion, execution time is

$$\begin{aligned}\text{CPU time} &= \# \text{ instructions} \times \text{CPI} \times \text{cycle time} \\ &= (100 \times 10^9) \times 1.15 \times (550 \times 10^{-12} \text{ sec}) \\ &= 63 \text{ seconds}\end{aligned}$$

- When we ran the same code on a single-cycle processor, CPU time was 92.5 seconds.

$$\text{Speedup of pipelined architecture} = 92.5 / 63 = 1.47x$$

Other Important Points

- **You do not have to know everything in the textbook. However, any content we have covered during the lecture so far could appear in the exam. If you have followed all our lectures, your hard work will pay off!**
- **Check the exam related info in the exam Module on Canvas.**
- **Good luck!**

SAN JOSÉ STATE UNIVERSITY *powering* SILICON VALLEY

