CS2100 Notes Part 2

L19: Sequential Logic

- Two types of sequential circuits:
  - Synchronous: outputs change only at specific time
  - Asynchronous: outputs change at any time
- Multivibrator: a class of sequential circuits
  - Bistable (2 stable states)
  - Monostable or one-shot (1 stable state)
  - Astable (no stable state)
- Bistable logic devices
  - Latches and flip-flops → differ in the methods used for changing their state

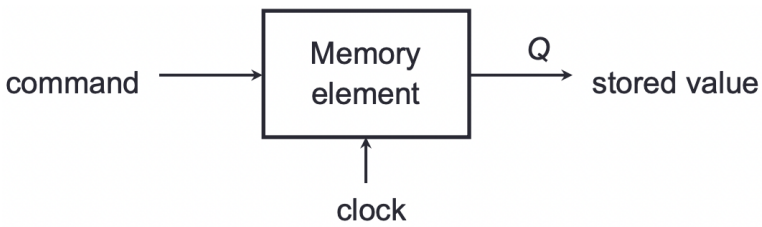
Memory Elements

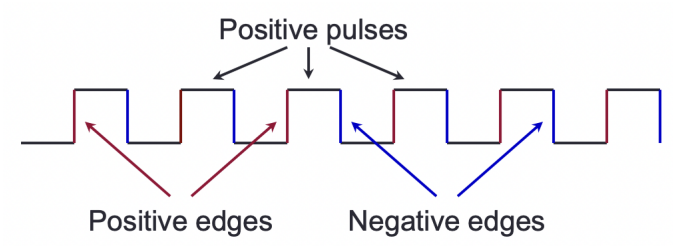
- a device which can remember value indefinitely, or change value on command from its inputs.

Command (at time *t*)	*Q(t)*	*Q(t+1)*
Set	X	1
Reset	X	0
Memorise / No Change	0	0
	1	1

Q(t) or ***Q***: current state

Q(t+1) or ***Q*⁺**: next state



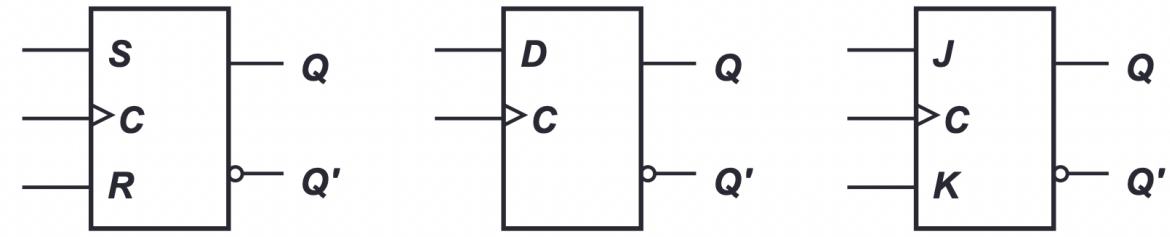


clock is usually a square wave

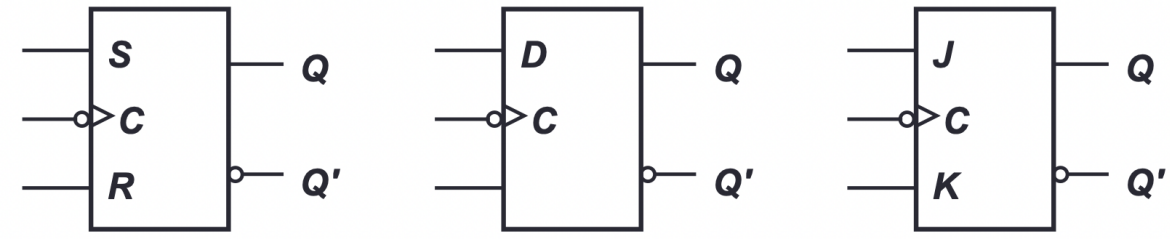
2 types of triggering/activation

- pulse-triggered → Latches → ON = 1, OFF = 0
- edge-triggered → clock
  - Flip-flops
  - Positive edge-triggered (ON = from 0 to 1; OFF = other time)
  - Negative edge-triggered (ON = from 1 to 0; OFF = other time)

Flip-Flops



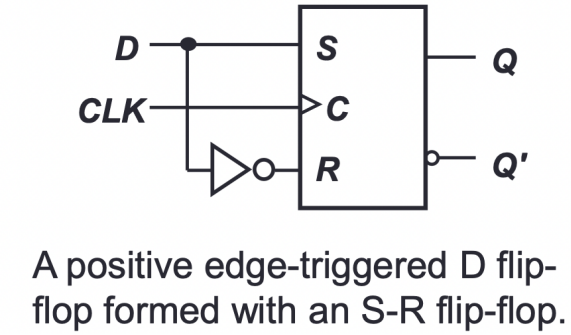
Positive edge-triggered flip-flops



Negative edge-triggered flip-flops

> rep clock

Convert S-R Flip-Flop into a D flip-flop



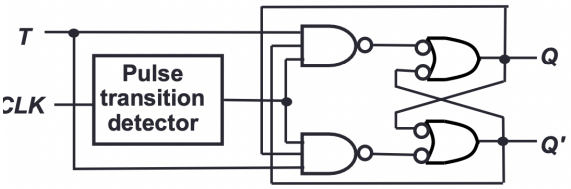
A positive edge-triggered D flip-flop formed with an S-R flip-flop.

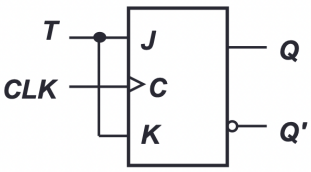
D	CLK	Q(t+1)	Comments
1	↑	1	Set
0	↑	0	Reset

↑ = clock transition LOW to HIGH

T Flip-Flop

- Single input version of the *J-K* flip-flop, formed by tying both inputs together.





Q	T	Q(t+1)
0	0	0
0	1	1
1	0	1
1	1	0

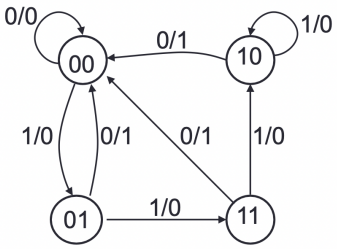
Asynchronous Inputs

- When *PRE* = HIGH, *Q* is immediately set to HIGH.
- When *CLR* = HIGH, *Q* is immediately cleared to LOW.
- Flip-flop in normal operation mode when both *PRE* and *CLR* are LOW.

Circuit Analysis

- *m* flip-flops → up to 2^m states.

Example 1



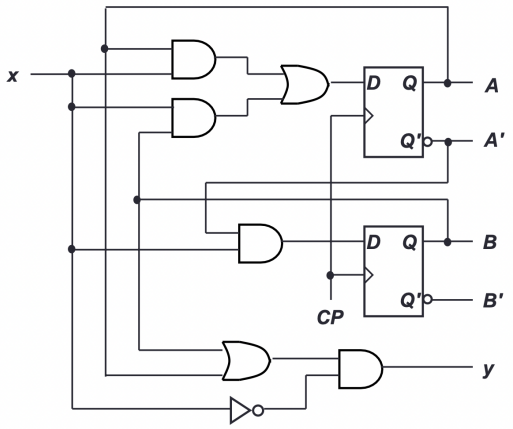
State equations:

$A^+ = A \cdot x + B \cdot x$

$B^+ = A' \cdot x$

Output function:

$y = (A + B) \cdot x'$



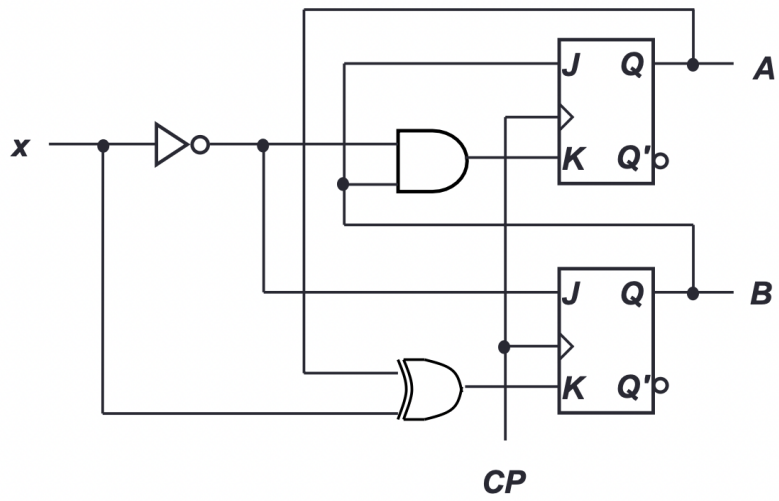
$A+ = A \cdot x + B \cdot x$ (for D: $Q+ = D$)

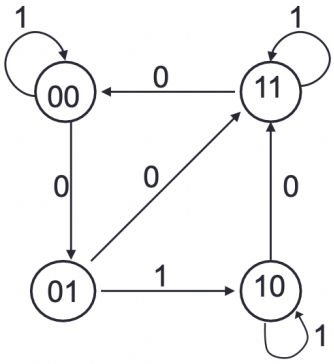
$B+ = A' \cdot x$

Present State		Input	Next State		Output
A	B	x	A^+	B^+	y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

Present State	Next State		Output	
	x=0	x=1	x=0	x=1
AB	A^+B^+	A^+B^+	y	y
00	00	01	0	0
01	00	11	1	0
10	00	10	1	0
11	00	10	1	0

Example 2





$JA = B$ $\quad\quad$ $JB = x'$

$KA = B \cdot x'$ $\quad\quad$ $KB = A' \cdot x + A \cdot x' = A \oplus x$

Fill the state table using the above functions, knowing the characteristics of the flip-flops used.

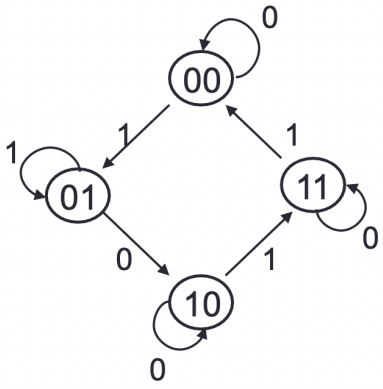
J	K	Q(t+1)	Comments
0	0	Q(t)	No change
0	1	0	Reset
1	0	1	Set
1	1	Q(t)'	Toggle

Present state		Input	Next state		Flip-flop inputs			
A	B	x	A^+	B^+	JA	KA	JB	KB
0	0	0	0	1	0	0	1	0
0	0	1	0	0	0	0	0	1
0	1	0	1	1	1	1	1	0
0	1	1	1	0	1	0	0	1
1	0	0	1	1	0	0	1	1
1	0	1	1	0	0	0	0	0
1	1	0	0	0	1	1	1	1
1	1	1	1	1	1	0	0	0

Circuit Design

- unused states → don't care values

- Circuit state/excitation table, using *JK* flip-flops.

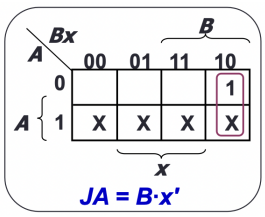


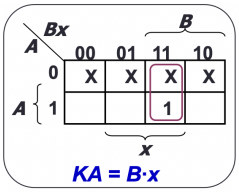
Q	Q^+	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

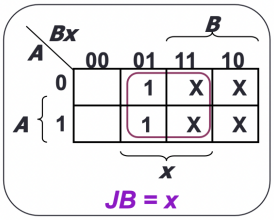
JK Flip-flop's excitation table.

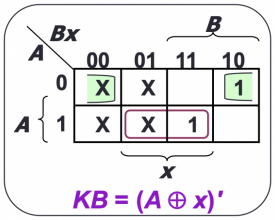
Present State	Next State x=0	Next State x=1
AB	A^+B^+	A^+B^+
00	00	01
01	10	01
10	10	11
11	11	00

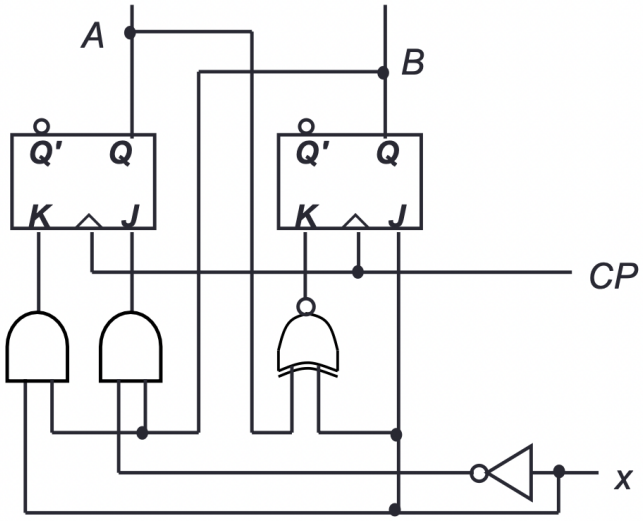
Present state		Input	Next state		Flip-flop inputs			
A	B	x	A^+	B^+	JA	KA	JB	KB
0	0	0	0	0	0	X	0	X
0	0	1	0	1	0	X	1	X
0	1	0	1	0	1	X	X	1
0	1	1	0	1	0	X	X	0
1	0	0	1	0	X	0	0	X
1	0	1	1	1	X	0	1	X
1	1	0	1	1	X	0	X	0
1	1	1	0	0	X	1	X	1











Flip-Flop SOP

Flip-Flop SOP

Aa Name	SOP (active high)	Note
S-R	***Q(t+1) = S + R'·Q***	***S·R*=0**
D	***Q(t+1) = D***	
J-K	***Q(t+1) = J·Q' + K'·Q***	
T	***Q(t+1) = T·Q' + T'·Q***	

Flip-Flop Characteristic Table (Analysis)

J	K	Q(t+1)	Comments
0	0	Q(t)	No change
0	1	0	Reset
1	0	1	Set
1	1	Q(t)'	Toggle

S	R	Q(t+1)	Comments
0	0	Q(t)	No change
0	1	0	Reset
1	0	1	Set
1	1	?	Unpredictable

D	Q(t+1)	
0	0	Reset
1	1	Set

T	Q(t+1)	
0	Q(t)	No change
1	Q(t)'	Toggle

given the flip-flop states, determine the next state; set = HIGH, reset = LOW

Flip-Flop Excitation Table (Design)

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

JK Flip-flop

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

SR Flip-flop

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

D Flip-flop

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

T Flip-flop

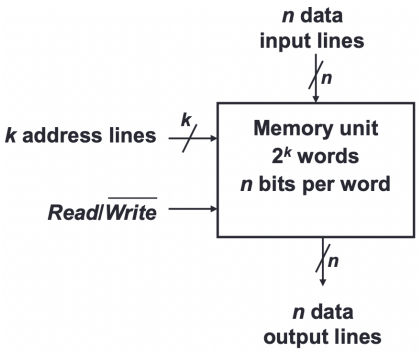
given the required transition from present state to next state, determine the flip-flop input(s).

Memory

- 1 byte = 8 bits

- 1 KB = 2^{10} bytes
- 1 MB = 2^{20} bytes
- 1 GB = 2^{30} bytes
- 1 TB = 2^{40} bytes

- A memory unit stores binary information in groups of bits called *words*.
- The data consists of *n* lines (for *n* bit words).
- Data input lines provide the information to be stored (*written*) into the memory, while data output lines carry the information out (*read*) from the memory.
- The address consists of *k* lines which specify which word (among the 2^*k* words available) to be selected for reading or writing.
- The control lines *Read* and *Write* (usually combined into a single control line *Read/Write*) specifies the direction of transfer of the data.



Memory Enable	*Read/Write*	Memory Operation
0	X	None
1	0	Write to selected word
1	1	Read from selected word

L20-21: Pipelining

Five execution stages - **Each execution stage takes 1 clock cycle** (except updating PC & write back of reg file):

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

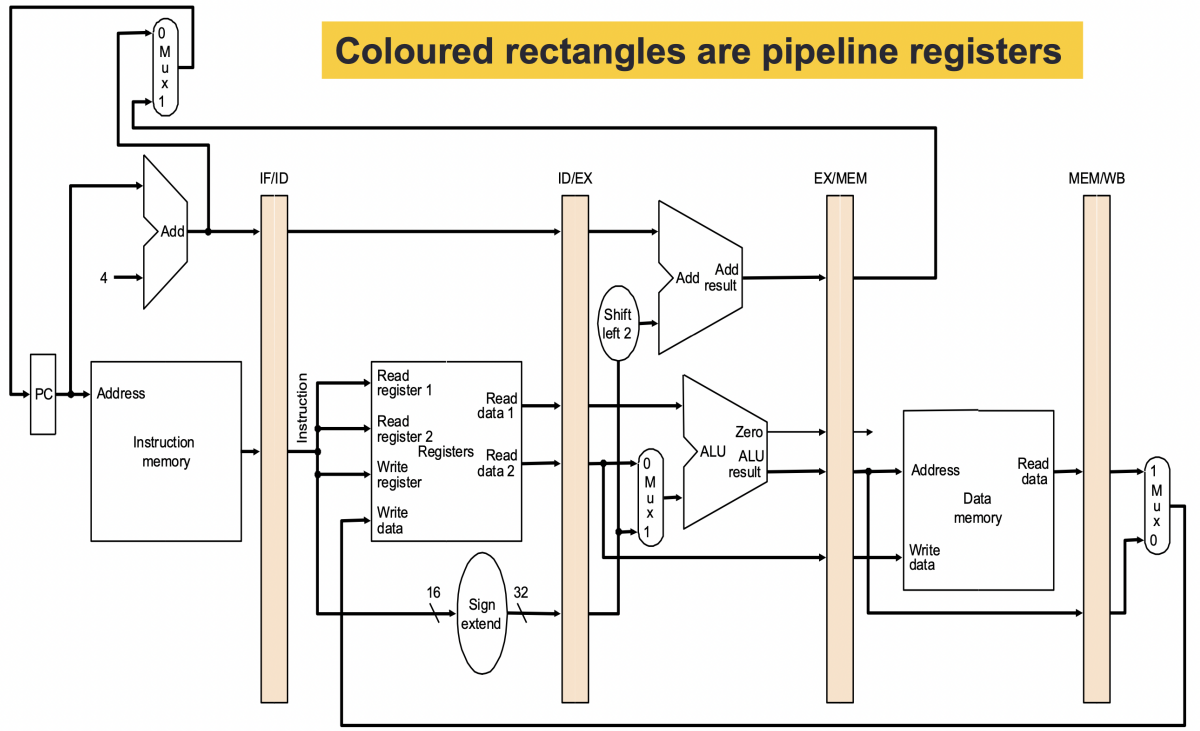
Pipelined Datapath

Single-cycle implementation:

- Update all state elements (**PC**, register file, data memory) at the end of a clock cycle

Pipelined implementation:

- One cycle per pipeline stage
- Data required for each stage needs to be stored separately
- Date used by **subsequent instructions:**
  - Store in programmer-visible state elements: **PC**, register file and memory
- Data used by **same instruction** in later pipeline stages:
  - Additional registers in datapath called **pipeline registers**
  - **IF/ID:** register between **IF** and **ID**
  - **ID/EX:** register between **ID** and **EX**
  - **EX/MEM:** register between **EX** and **MEM**
  - **MEM/WB:** register between **MEM** and **WB**

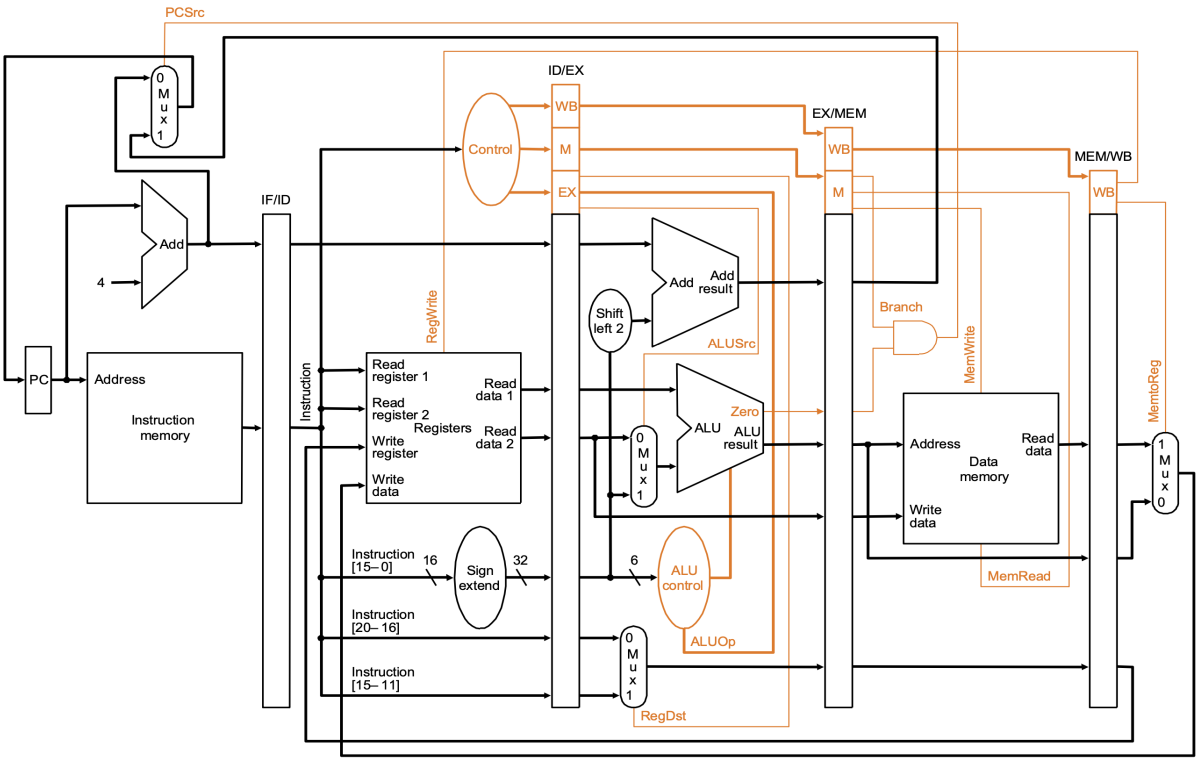


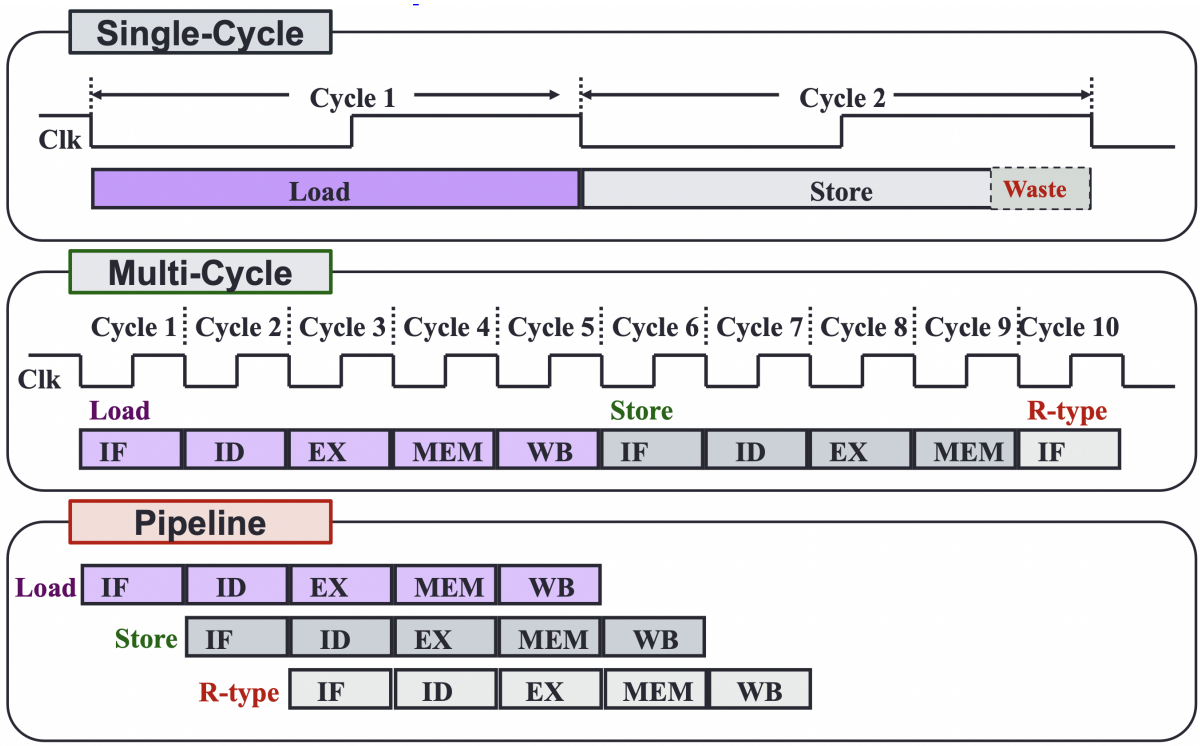
Pipeline Stages

Aa Stage	Start of cycle	End of cycle
IF		IF/ID reg receives (stores): ❖ Instruction read from InstructionMemory[PC] ❖ PC + 4 (Also connected to one of the MUX's inputs (another coming later))
ID	IF/ID supplies: ❖ Register numbers for reading two registers ❖ 16-bit offset to be sign-extended to 32-bit	ID/EX receives: ❖ Data values read from register file ❖ 32-bit immediate value ❖ PC + 4
EX	ID/EX supplies: ❖ Data values read from register file ❖ 32-bit immediate value ❖ PC + 4	EX/MEM receives: ❖ (PC + 4) + (Immediate x 4) ❖ ALU result ❖ `isZero?` signal ❖ Data Read 2 from register file
MEM	EX/MEM supplies: ❖ (PC + 4) + (Immediate x 4) ❖ ALU result ❖ isZero? signal ❖ Data Read 2 from register file	MEM/WB receives: ❖ ALU result ❖ Memory read data
WB	MEM/WB supplies: ❖ ALU result ❖ Memory read data	❖ Result is written back to register file (if applicable)

Pipeline Control

	EX Stage				MEM Stage			WB Stage	
	RegDst	ALUSrc	ALUop		Mem Read	Mem Write	Branch	MemTo Reg	Reg Write
			op1	op0					
R-type	1	0	1	0	0	0	0	0	1
lw	0	1	0	0	1	0	0	1	1
sw	X	1	0	0	0	1	0	X	0
beq	X	0	0	1	0	0	1	X	0





Single-Cycle Processor

- Cycle time: $CT_{seq} = \max(\sum_{k=1}^{N} T_k)$
  - T_k = Time for operation in stage k
  - N = Number of stages
- Execution Time for `I` instructions:
  - $Time_{seq} = I \times CT_{seq}$

Instruction	Inst Mem	Reg read	ALU	Data Mem	Reg write	Total
ALU (eg: add)	2	1	2		1	6
lw	2	1	2	2	1	8
sw	2	1	2	2		7
beq	2	1	2			5

Assume 100 instructions.

Cycle time = 8ns

Time to execute 100 instructions =100×8ns=800ns

Multi-Cycle Processor

- Cycle time: $CT_{multi} = \max(T_k)$
- Execution Time for `I` instructions:
  - $Time_{multi} = I \times Average\ CPI \times CT_{multi}$
  - Average CPI needed as each instruction takes different number of cycles

Cycle time = 2ns

Given average CPI = 4.6

Time to execute 100 instructions = 100 × 4.6 × 2ns = 920ns

CPI is calculated from stat analysis

Pipelining Processor

- Cycle time: $CT_{pipeline} = \max(T_k) + T_d$
  - T_d = Overhead for pipelining, e.g. pipeline register
- Cycles needed for `I` instructions:
  - $I + N - 1$ (need $N - 1$ cycles to fill up the pipeline)
- Execution Time for `I` instructions:
  - $Time_{pipeline} = (I + N - 1) \times CT_{pipeline}$

Assume pipeline register latency = 0.5ns

Cycle time = 2ns + 0.5ns = 2.5ns

Time to execute 100 instructions = (100+5–1) × 2.5ns = 260ns.

- Assumptions for ideal case:
  - Every stage takes the same amount of time → $\sum_{k=1}^{N} T_k = N \times T_1$
  - No pipeline overhead → $T_d = 0$
  - $I >> N$ (Number of instructions is much larger than number of stages)
- $Speedup_{pipeline} = \frac{Time_{seq}}{Time_{pipeline}}$

$$= \frac{I \times \sum_{k=1}^{N} T_k}{(I+N-1) \times (\max(T_k)+T_d)} = \frac{I \times N \times T_1}{(I+N-1) \times T_1}$$

$$\approx \frac{I \times N \times T_1}{I \times T_1}$$

$$= \boldsymbol{N}$$

Conclusion:
Pipeline processor can gain **N** times speedup, where **N** is the number of pipeline stages.

Exercise:

```
add $t0, $s0, $s1
sub $t1, $s0, $s1
sll $t2, $s0, 2
srl $t3, $s1, 2
```

(a) How many cycles will it take to execute the code on a single-cycle datapath?

4 cycles.

(b) How long will it take to execute the code on a single-cycle datapath, assuming a 100 MHz clock?

$4 \div (100 \times 10^6) = 40ns$

(c) How many cycles will it take to execute the code on a 5-stage MIPS pipeline?

4 + (5 - 1) = 8

ideal: number of instructions + (num cycles(i.e. stages) - 1)

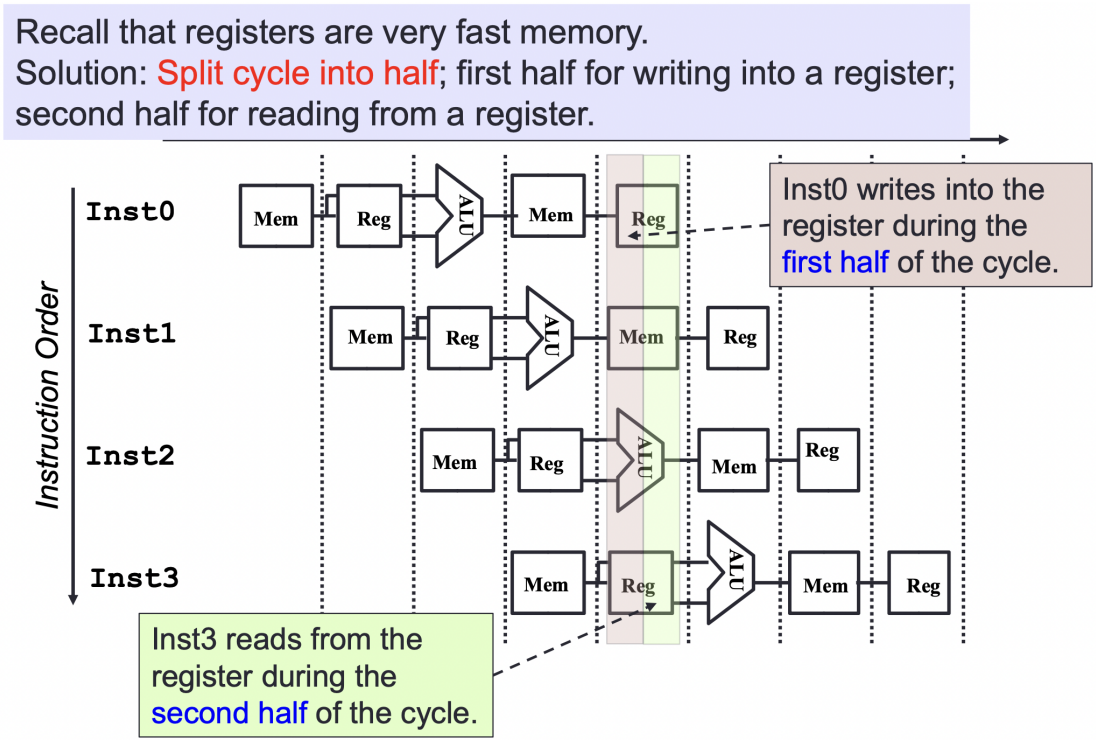
(d) How long will it take to execute the code on a 5-stage MIPS pipeline, assuming a 500 MHz clock?

$8 \div (500 \times 10^6) = 16ns$

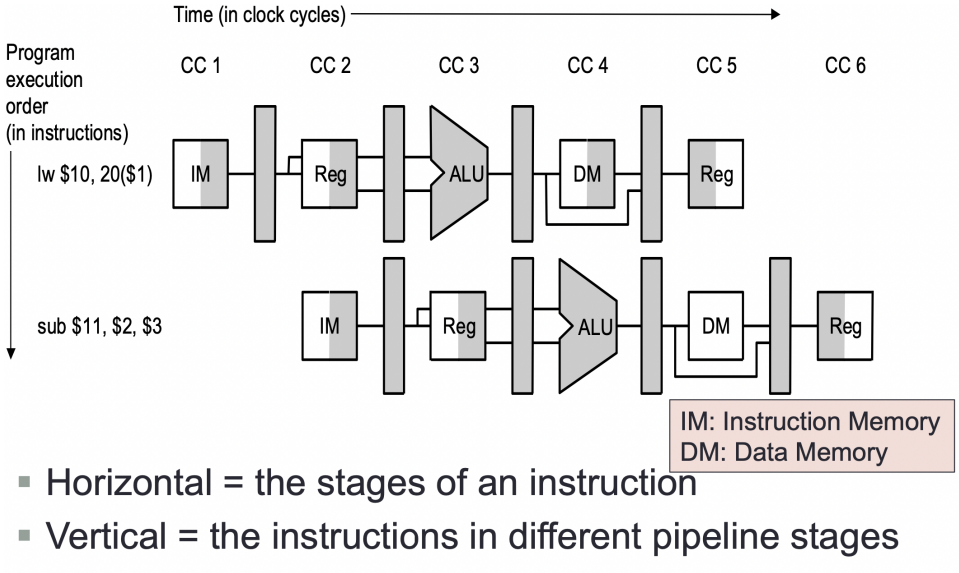
Pipeline Hazards

Problems that prevent next instruction from immediately following previous instruction

- **Structural hazards:** Simultaneous use of a hardware resource (e.g. writing to MEM simultaneously for diff instructions)
  - Solution 1: Stall the pipeline
  - Split memory into **Data** and **Instruction** memory



- **Data hazards:** Data dependencies between instructions
- **Control hazards:** Change in program flow



Instruction Dependencies

- When different instructions accesses (read/write) the same register
  - Register contention is the cause of dependency
  - Known as **data dependency**
- When the execution of an instruction depends on another instruction
  - Control flow is the cause of dependency
  - Known as **control dependency**

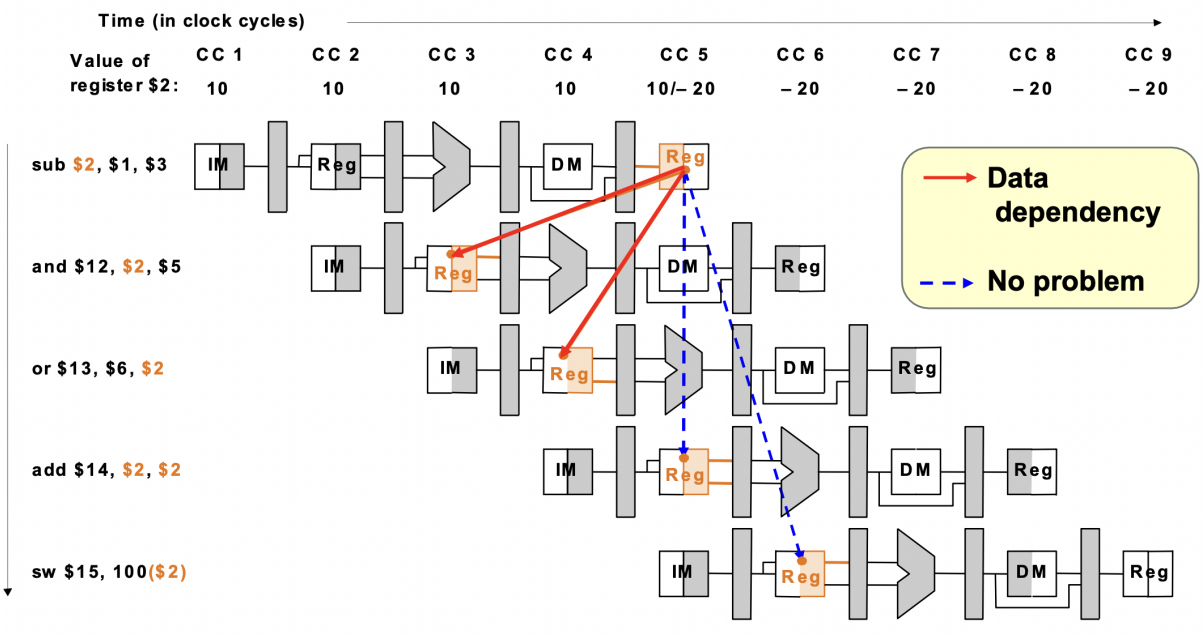
Data Dependency

Read-After-Write (RAW)

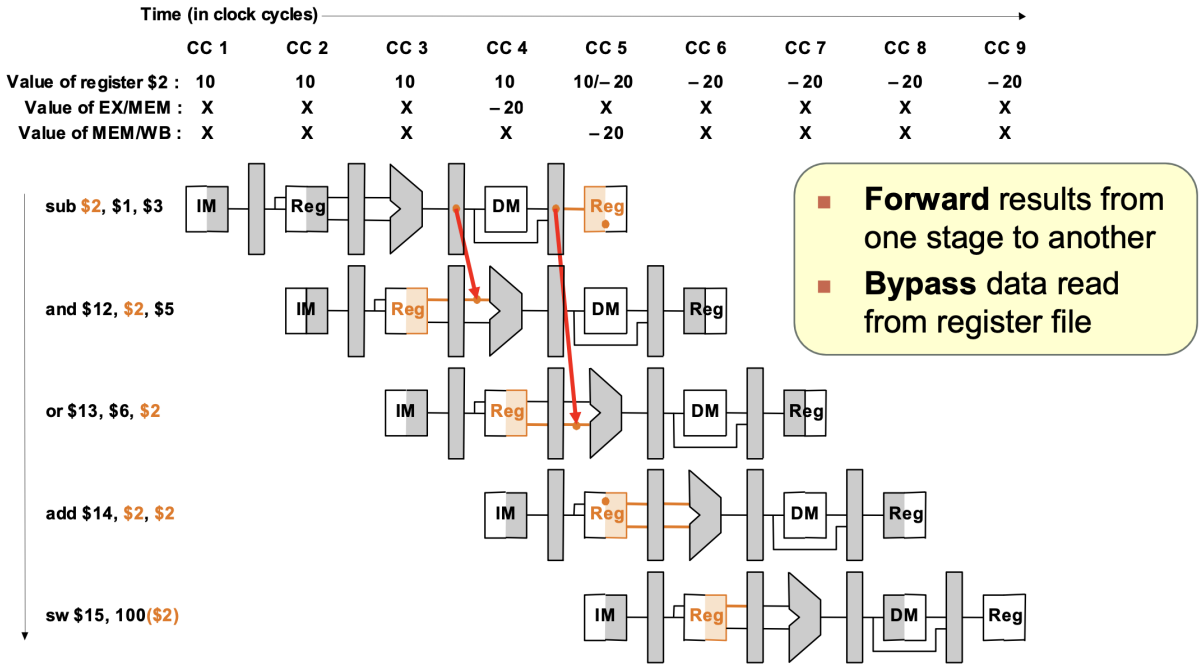
- Occurs when a later instruction **reads** from the destination register **written** by an earlier instruction
- Aka **true data dependency**

```
i1: add $1, $2, $3 #writes to $1
i2: sub $4, $1, $5 #reads from $1
```

If **i2** reads register **$1** before **i1** can write back the result, **i2** will get a ***stale result (old result)***



Forwarding



- `load` cannot resolve with forwarding as the data is only available after MEM stage

```
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
```

	1	2	3	4	5	6	7	8	9	10	11
sub	IF	ID	EX	MEM	WB						
and		IF			ID	EX	MEM	WB			
or					IF	ID	EX	MEM	WB		
add						IF	ID	EX	MEM	WB	
sw							IF	ID	EX	MEM	WB

without forwarding

	1	2	3	4	5	6	7	8	9	10	11
sub	IF	ID	EX	MEM	WB						
and		IF	ID	EX	MEM	WB					
or			IF	ID	EX	MEM	WB				
add				IF	ID	EX	MEM	WB			
sw					IF	ID	EX	MEM	WB		

with forwarding

```
lw $2, 20($3)
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
```

	1	2	3	4	5	6	7	8	9	10	11
lw	IF	ID	EX	MEM	WB						
and		IF			ID	EX	MEM	WB			
or					IF	ID	EX	MEM	WB		
add						IF	ID	EX	MEM	WB	
sw							IF	ID	EX	MEM	WB

	1	2	3	4	5	6	7	8	9	10	11
lw	IF	ID	EX	MEM	WB						
and		IF	ID		EX	MEM	WB				
or			IF		ID	EX	MEM	WB			
add					IF	ID	EX	MEM	WB		
sw						IF	ID	EX	MEM	WB	

Control Dependency

- An instruction **j** is control dependent on **i** if **i** controls whether or not **j** executes
- Typically **i** would be a branch instruction

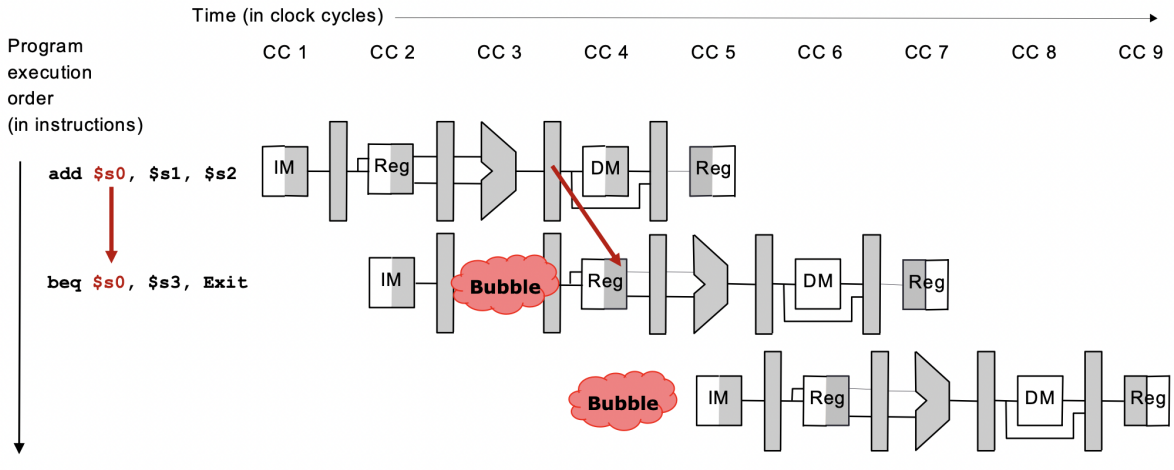
```
i1: beq $3, $5, label    # branch
i2: add $1, $2, $4       # depends on i1
...   ...   ...
```

If **i2** is allowed to execute before **i1** is determined, register **$1** maybe incorrectly changed!

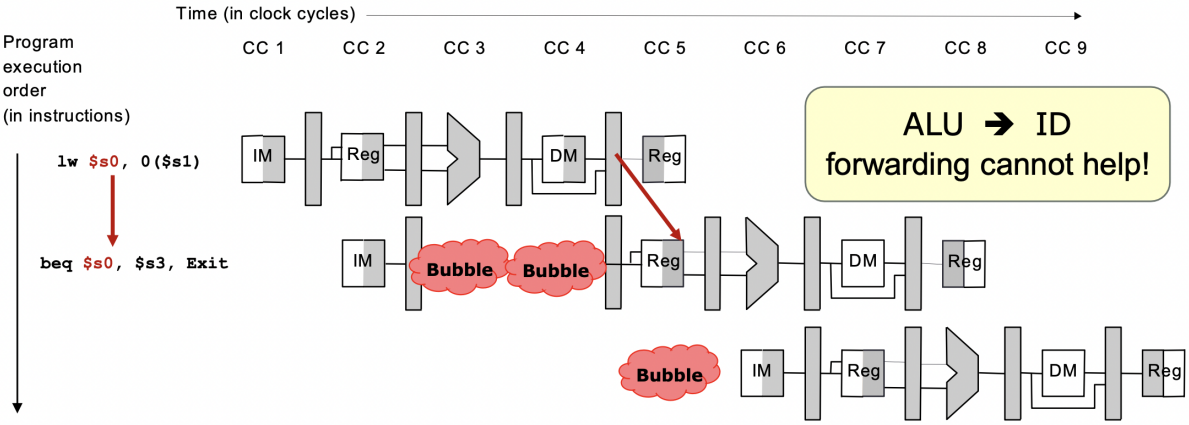
Early Branch Resolution

- Move branch decision calculation to earlier pipeline stage

- Make decision in **ID** stage instead of **MEM (need have data at ID stage)**
  - Move branch target address calculation
  - Move register comparison
  - cannot use ALU for register comparison any more



Add forwarding path from ALU to ID stage → **One clock cycle delay** is still needed



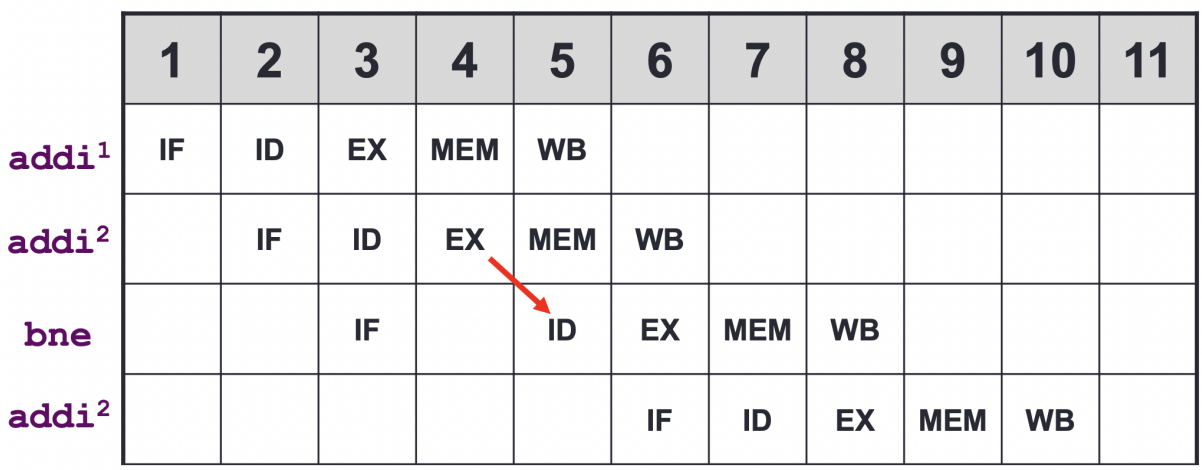
`load` followed by branch → data only avail after M, need more stalling → same 3 total stall cycles

Branch Prediction

- Guess the outcome before it is produced
- All branches are assumed to be **not taken**
- **Fetch the successor instruction and start pumping it through the pipeline stages**
- Wrong prediction → **Flush** successor instruction from the pipeline

Exercise:

```
      addi $s0, $zero, 10
Loop: addi $s0, $s0, -1
      bne  $s0, $zero, Loop # data dependency -> 1 cycle of delay
      sub  $t0, $t1, $t2
# Decision making moved to ID stage
```



- Total instructions = 1 + 10 x 2 + 1 = 22
- **Ideal** pipeline = 4 + 22 = 26 cycles

Without branch prediction:

- Data dependency between (**addi $s0, $s0, -1**) and **bne** incurs 1 cycle of delay. 10 iterations → 10 cycles of delay.
- Every **bne** incurs a cycle of delay to execute the next instruction. 10 iterations → 10 cycles of delay.
- Total number of cycles of delay = 20.
- Total execution cycles = 26 + 20 = **46 cycles.**

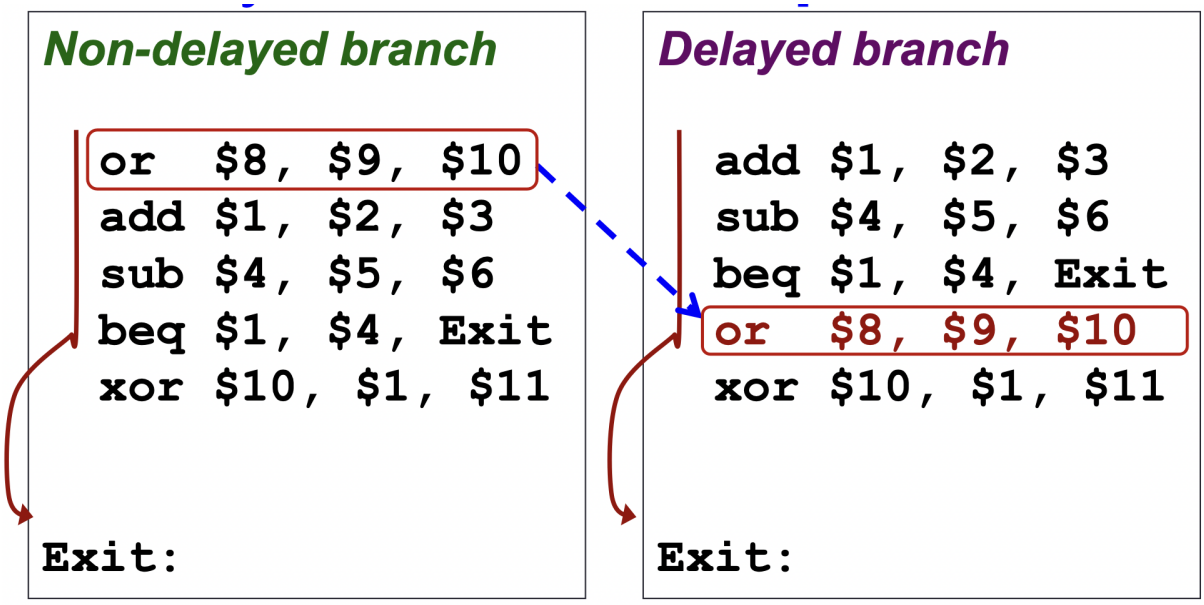
	1	2	3	4	5	6	7	8	9	10	11
addi[1]	IF	ID	EX	MEM	WB						
addi[2]		IF	ID	EX	MEM	WB					
bne			IF		ID	EX	MEM	WB			
sub					IF						
addi[2]						IF	ID	EX	MEM	WB	

With branch prediction:

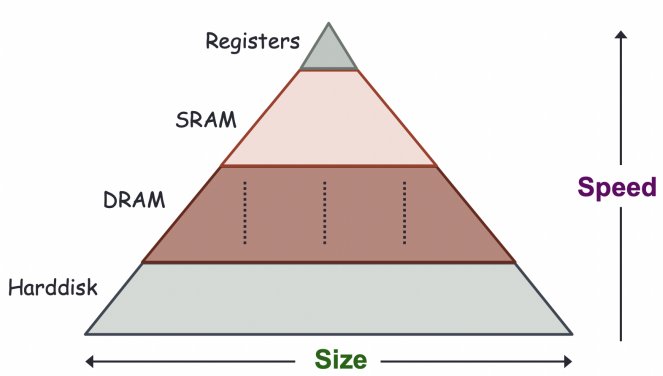
- data dependency: 10 cycles of delay for 10 iterations.
- In the first 9 iterations, the branch prediction is wrong, hence 1 cycle of delay.
- In the last iteration, the branch prediction is correct, hence saving 1 cycle of delay.
- Total number of cycles of delay = 19.
- Total execution cycles = 26 + 19 = **45 cycles.**

Delayed Branching

- Move **non-control dependent instructions** into the **slots following a branch**
  - Known as the **branch-delay slot**
- These instructions are executed regardless of the branch outcome
- **Worst case scenario**
  - no such instruction → Add a no-op (nop) instruction in the branch-delay slot



L22-23: Cache



Terminology

- **Hit**: Data is in cache (e.g., **X**)
  - Hit rate: Fraction of memory accesses that hit
  - Hit time: Time to access cache
- **Miss**: Data is not in cache (e.g., **Y**)
  - Miss rate = 1 – Hit rate
  - Miss penalty: Time to replace cache block + hit time

Formula

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

Example

▼ Suppose our on-chip SRAM (cache) has **0.8 ns** access time, but the fastest DRAM (main memory) we can get has an access time of **10ns**. **How high a hit rate** do we need to sustain an average access time of **1ns**?

Let h be the desired hit rate.

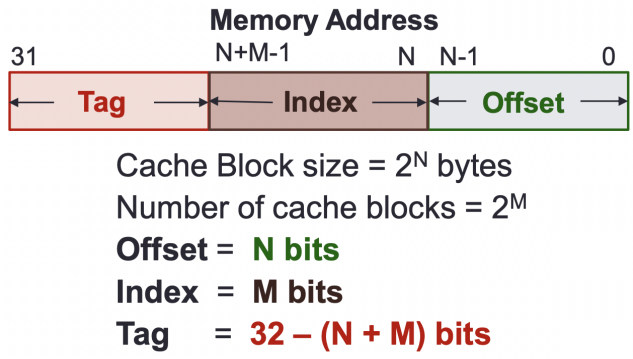
$1 = 0.8h + (1 - h) \times (10 + 0.8)$

$= 0.8h + 10.8 - 10.8h$

$10h = 9.8 \rightarrow h = 0.98$

Hence we need a hit rate of **98%**.

Direct-Mapped Cache



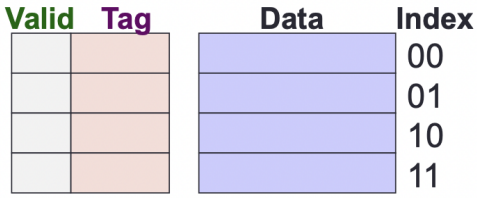
Cache Block size = 2^N bytes

Number of cache blocks = 2^M

Offset = **N bits**

Index = **M bits**

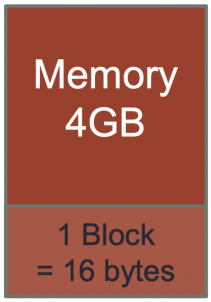
Tag = **32 – (N + M) bits**

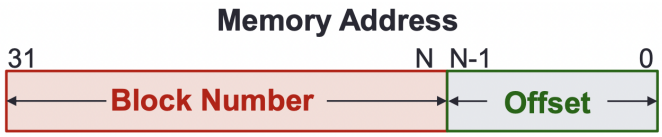


Cache Structure:

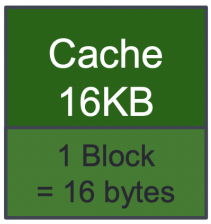
1. num of offset bits = log2(bytes in each block)
2. num of index bits = log2(cache capacity/block capacity)
3. tag = block num / num of cache blocks

Cache hit = (Valid[index] = TRUE) **AND** (Tag[index] = Tag[memory address])

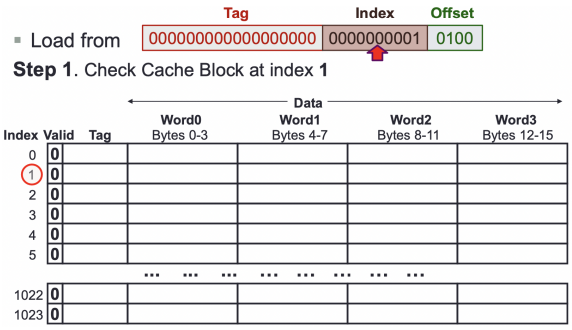


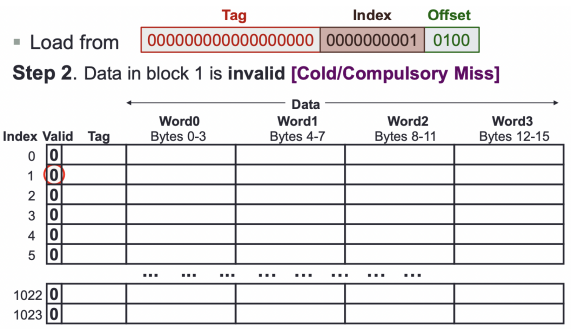


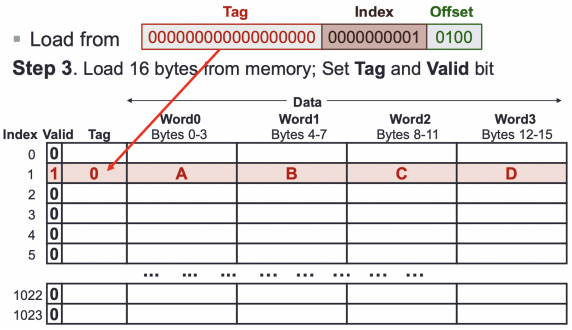
Offset, N = 4 bits
Block Number = 32 – 4 = **28 bits**
Check: Number of Blocks = 2^{28}

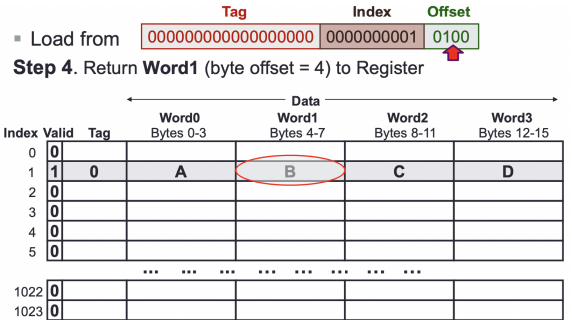


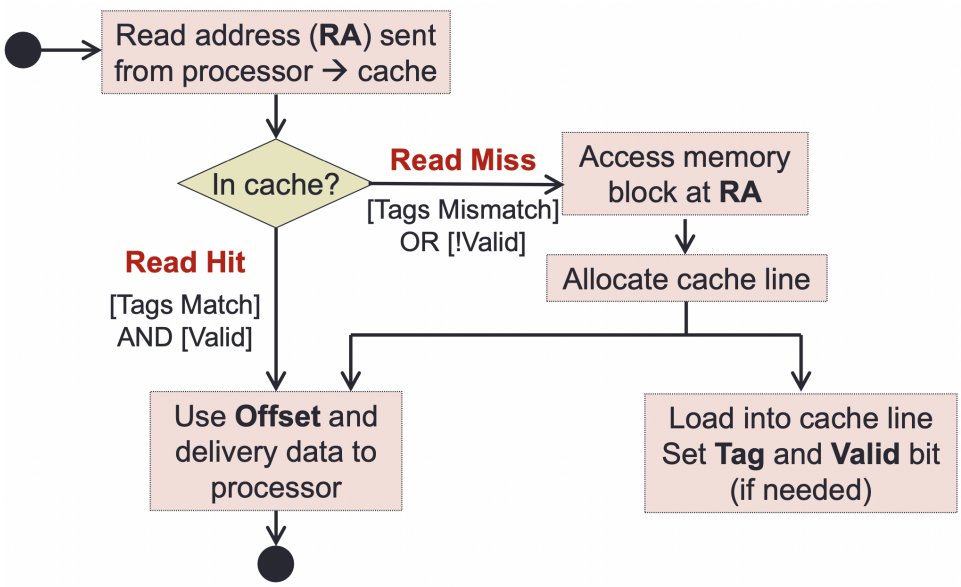
Number of Cache Blocks
= 16KB / 16bytes = 1024 = 2^{10}
Cache Index, M = 10bits
Cache Tag = 32 – 10 – 4 = **18 bits**





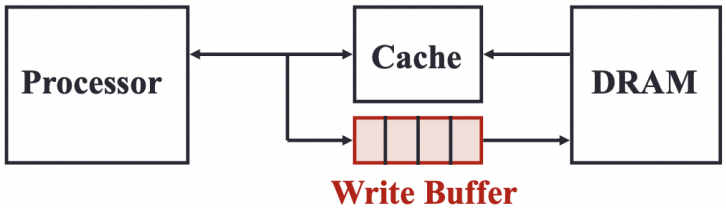






Write Policy

1. **Write-through** cache
    - Write data both to cache and to main memory
    - Slow → Solution: Put a write buffer between cache and main memory
        - Processor: writes data to cache + write buffer
        - Memory controller: write contents of the buffer to memory



2. **Write-back** cache
    - Only write to cache
    - Write to main memory only when cache block is replaced (evicted)
    - Wasteful if we write back every evicted cache blocks → Solution
        - Add an additional bit (**Dirty bit**) to each cache block
        - **Write operation will change dirty bit to 1**
            - Only cache block is updated, no write to memory
        - **When a cache block is replaced:**

- Only write back to memory if dirty bit is 1

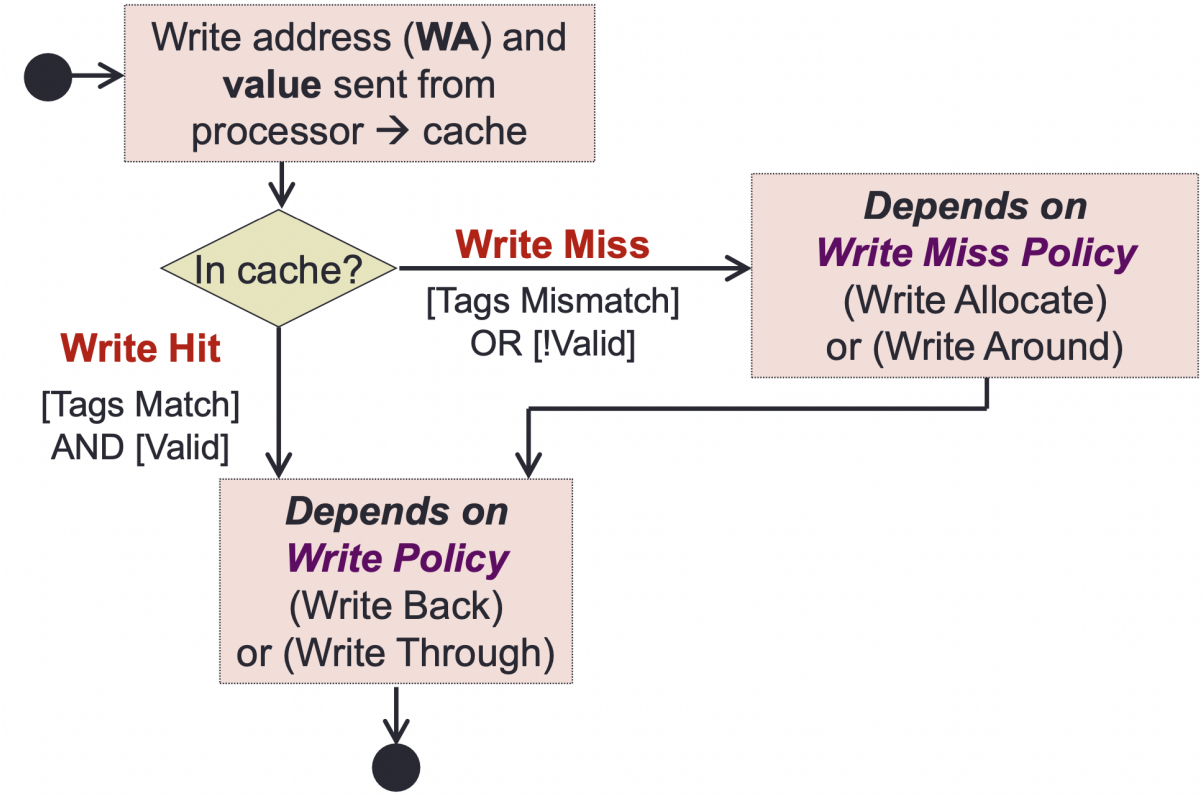
Write Miss Policy

Write Miss option 1: **Write allocate**

- Load the complete block into cache
- Change only the required word in cache
- Write to main memory depends on write policy

Write Miss option 2: **Write around**

- Do not load the block to cache
- Write directly to **main memory only**



Set Associative Cache

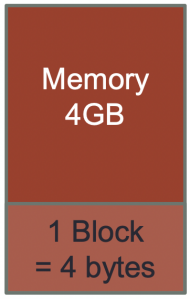
- Solve conflict misses
- **N-way Set Associative Cache**

- A memory block can be placed in a fixed number (N) of locations in the cache, where **N** > 1
- Cache consists of a number of sets
  - Each set contains **N** cache blocks
- Each memory block maps to a **unique** cache set
- Within the set, a memory block can be placed in **any** of the N cache blocks in the set → need search through

Cache Set Index
= (BlockNumber) modulo (NumberOfCacheSets)

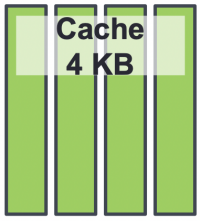
Cache Block size = 2^N bytes
Number of cache sets = 2^M
Offset = **N bits**
Set Index = **M bits**
Tag = **32 – (N + M) bits**

Observation:
It is essentially unchanged from the direct-mapping formula



Memory Address

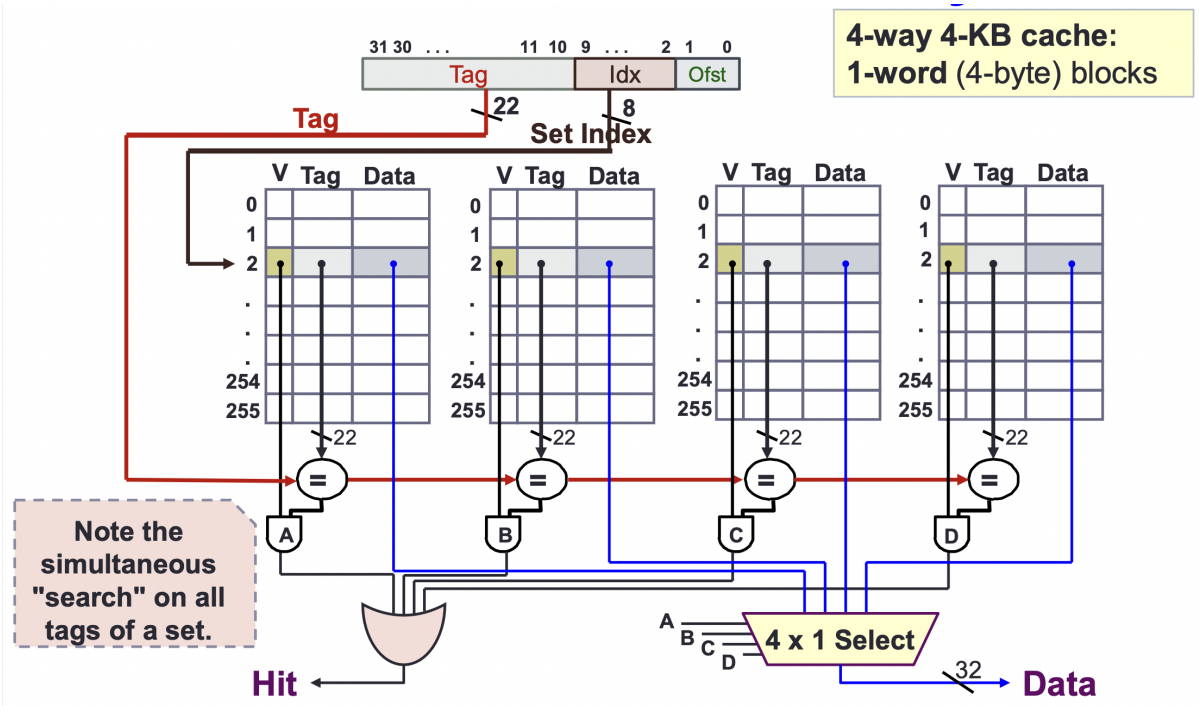
Offset, N = 2 bits
Block Number = 32 – 2 = **30 bits**
Check: Number of Blocks = 2^{30}



Number of Cache Blocks
= 4KB / 4bytes = 1024 **= 2^{10}**

4-way associative, number of sets
= 1024 / 4 = 256 **= 2^8**
Set Index, M = 8 bits

Cache Tag = 32 – 8 – 2 = **22 bits**



3. SA Cache Example: **Load #1**

Miss

4, 0, 8, 36, 0

- Load from **4** → Tag: 00000000000000000000000000 | Index: 0 | Offset: 100

Check: Both blocks in **Set 0** are invalid **[Cold Miss]**

Result: Load from memory and place in **Set 0 - Block 0**

Set Index	Block 0				Block 1			
	Valid	Tag	W0	W1	Valid	Tag	W0	W1
0	~~0~~ 1	0	M[0]	M[4]	0			
1	0				0			

3. SA Cache Example: **Load #2**

Miss Hit

4, 0, 8, 36, 0

- Load from **0** → Tag: 00000000000000000000000000 | Index: 0 | Offset: 000

Result:
[Valid and Tags match] in **Set 0-Block 0 [Spatial Locality]**

Set Index	Block 0				Block 1			
	Valid	Tag	W0	W1	Valid	Tag	W0	W1
0	1	0	M[0]	M[4]	0			
1	0				0			

3. SA Cache Example: **Load #3**

Miss Hit Miss

4, 0, 8, 36, 0

- Load from **8** → Tag: 00000000000000000000000000 | Index: 1 | Offset: 000

Check: Both blocks in **Set 1** are invalid **[Cold Miss]**

Result: Load from memory and place in **Set 1 - Block 0**

Set Index	Block 0				Block 1			
	Valid	Tag	W0	W1	Valid	Tag	W0	W1
0	1	0	M[0]	M[4]	0			
1	~~0~~ 1	0	M[8]	M[12]	0			

3. SA Cache Example: **Load #4**

Miss Hit Miss Miss

4, 0, 8, 36, 0

- Load from **36** → Tag: 00000000000000000000000010 | Index: 0 | Offset: 100

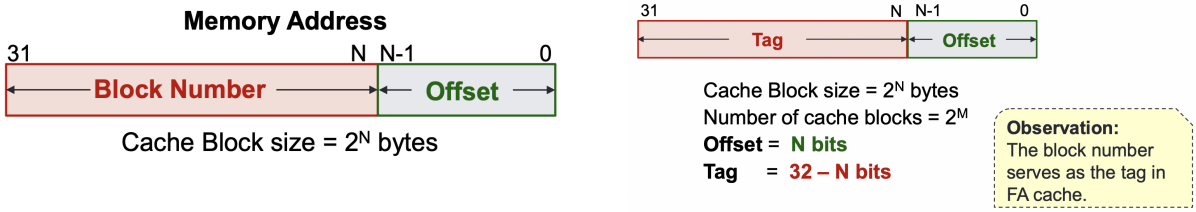
Check: [Valid but tag mismatch] **Set 0 - Block 0**
[Invalid] **Set 0 - Block1 [Cold Miss]**

Result: Load from memory and place in **Set 0 - Block 1**

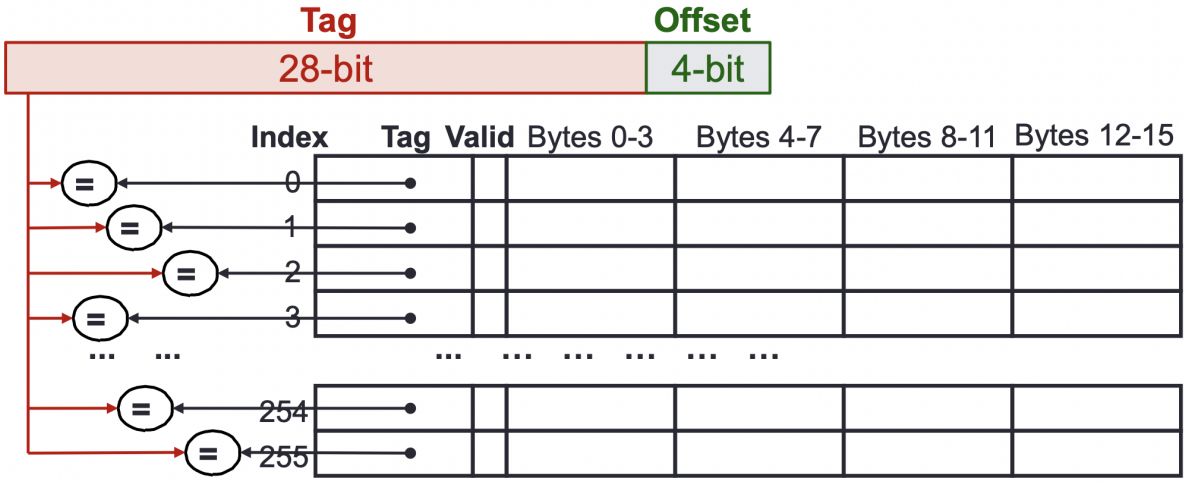
Set Index	Block 0				Block 1			
	Valid	Tag	W0	W1	Valid	Tag	W0	W1
0	1	0	M[0]	M[4]	~~0~~ 1	2	M[32]	M[36]
1	1	0	M[8]	M[12]	0			

Fully Associative Cache

- A memory block can be placed in any location in the cache
- Memory block placement is no longer restricted by cache index or cache set index
  - adv: can be placed in any location
  - disadv: Need to search all cache blocks for memory access



- 4KB cache size and 16-Byte block size
- Compare tags and valid bit in parallel



No Conflict Miss (since data can go anywhere)

Total Miss = Cold miss + Conflict miss + Capacity miss

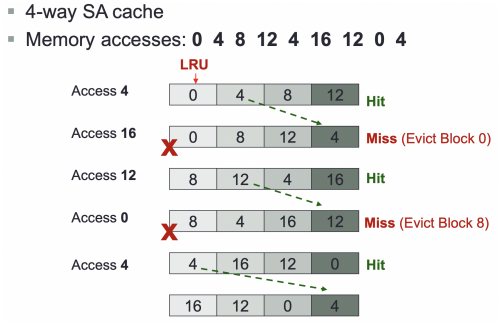
Capacity miss (FA) = Total miss (FA) – Cold miss (FA) (Conflict Miss = 0)

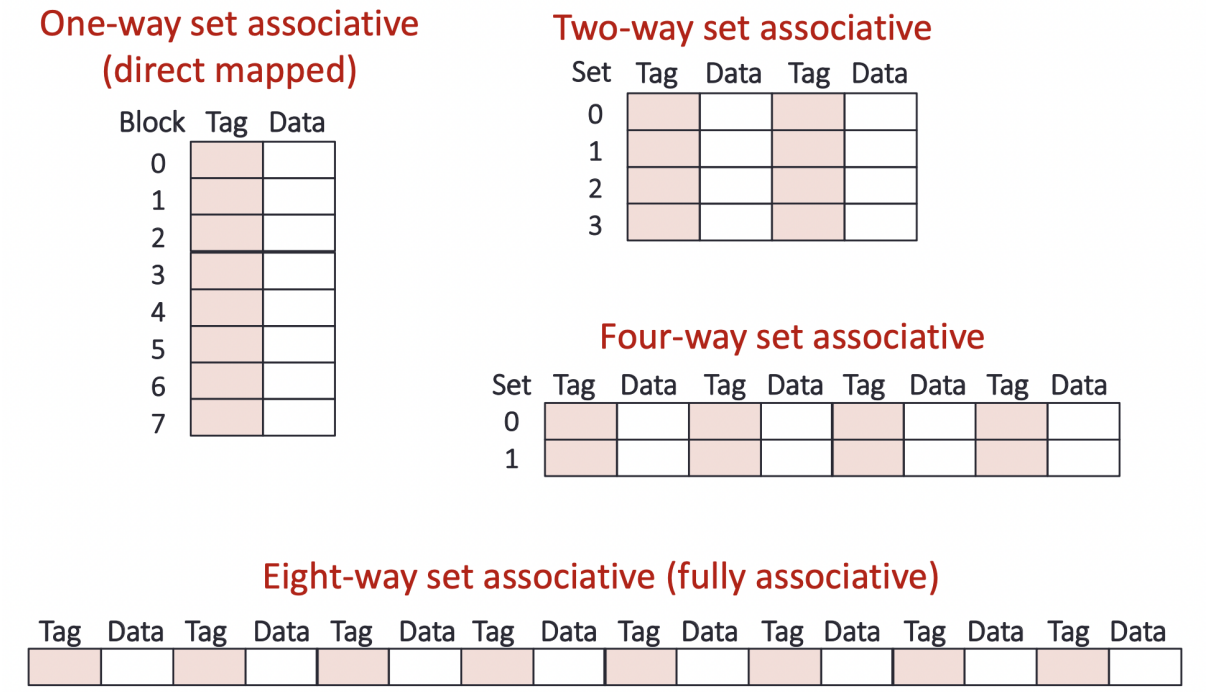
Block Replacement Policy

- For set associative or fully associative cache

Least **R**ecently Used (**LRU**)

- **How:** For cache hit, record the cache block that was accessed
  - When replacing a block, choose one which has not been accessed for the longest time
- **Why:** Temporal locality





Cache Framework Summary

Name	Block Placement	Block Identification	Block Replacement
Direct Mapped	Only 1 block defined by index	Tag match with only one block	No choice
N-way Set-Associative	Any one of the N blocks within the set defined by index	Tag match for all the blocks within the set	Based on replacement policy
Fully Associative (FA)	Any cache block	Tag match for all the blocks within the cache	Based on replacement policy

Name	Block Placement	Block Identification	Block Replacement
Write Strategy	Write Policy: Write-through vs write-back	Write Miss Policy: Write allocate vs write no allocate	
Average Access Time = Hit rate x Hit Time + (1-Hit rate) x Miss penalty			

Tutorial 1

(r - 1)'s complement

We generalise (*r* – 1)'s-complement (also called radix diminished complement) to include fraction as follows: **(*r*–1)'s complement of *N* =** $r^n - r^{-m} - N$

where *n* is the number of integer digits and *m* the number of fractional digits. (If there are no fractional digits, then *m* = 0 and the formula becomes $r^n - 1 - N$ as given in class.)

e.g. the 1's complement of 011.01 is $(2^3 - 2^{-2}) - 011.01 = (1000 - 0.01) - 011.01 = 111.11 - 011.01 = 100.10$

Tutorial 2

***x* AND 0 = 0**

x* AND 1 = *x

x* OR 0 = *x

***x* OR 1 = 1**

x* XOR 0 = *x

x* XOR 1 = *x'

▼ Set bits 2, 8, 9, 14 and 16 of **b** to 1. Leave all other bits unchanged.

To set bits, we create a "mask" with 1's in the bit positions we want to set.

```
Since bit 16 is in the upper 16 bits of the register,
we need to use lui to set it.

lui $t0, 1 # Sets bit 16 of $t0.
ori $t0, $t0, 0b0100001100000100 # Set bits 14, 9, 8 and 2.
or $s1, $s1, $t0
```

▼ Copy over bits 1, 3 and 7 of **b** into **a**, without changing any other bits of **a**.

```
# We use the property that x AND 1 = x to copy out
# the values of bits 7, 3 and 1 of b into $t0.
andi $t0, $s1, 0b0000000010001010

# We use the property of x OR 0 = x to copy in the bits into a,
# so we prepare a by zero-ing bits 7, 3 and 1.
lui $t1, 0b1111111111111111
ori $t1, $t1, 0b1111111101110101
and $s0, $s0, $t1

# Now OR together a and $t0 to copy over the bits
or $s0, $s0, $t0
```

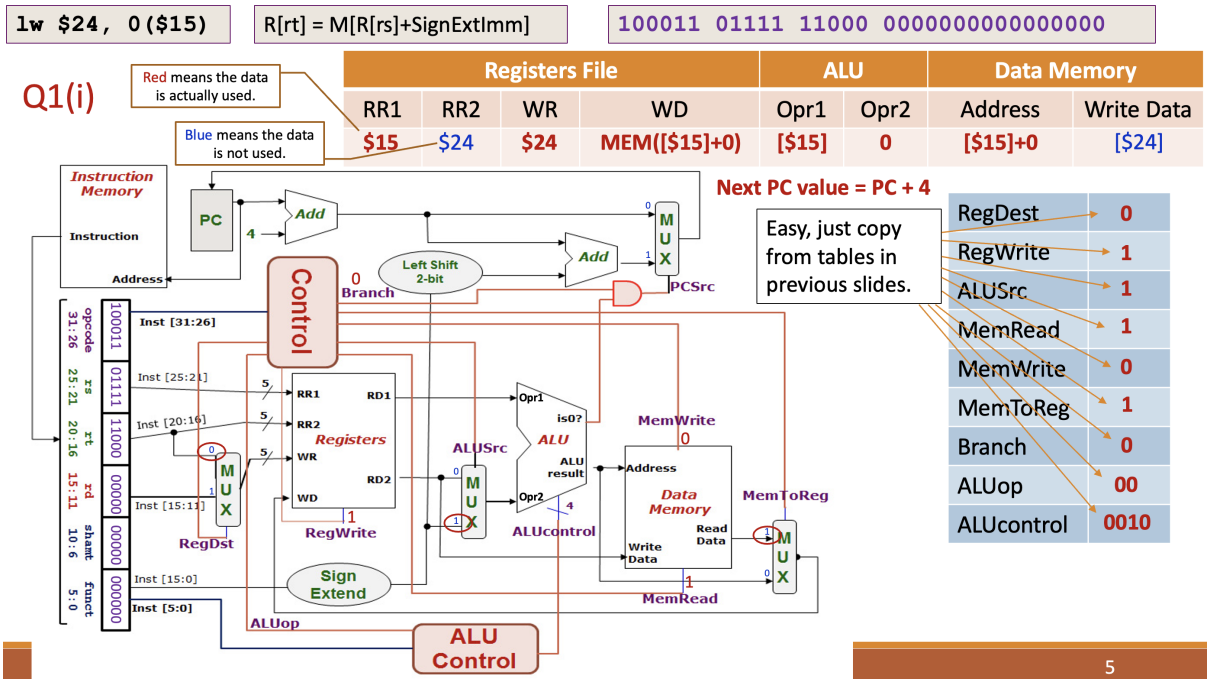
▼ Make bits 2, 4 and 8 of **c** the inverse of bits 1, 3 and 7 of **b** (i.e. if bit 1 of **b** is 0, then bit 2 of **c**
should be 1; if bit 1 of **b** is 1, then bit 2 of **c** should be 0), without changing any other bits of **c**.

```
# We use the property that x XOR 1 = ~x to flip the values of bits 7, 3 and 1.
xori $t0, $s1, 0b10001010

# Zero every bit except 7, 3 and 1.
andi $t0, $t0, 0b10001010

# Shift left one position:
# bit 1 becomes 0, bit 1 becomes bit 2, bit 3 becomes bit 4, and
# bit 7 becomes bit 8.
sll $t0, $t0, 1

# Now, to clear bits 8, 4 and 2 of c,
# we need the mask 0b1111111111111111 1111111011101010
lui $t1, 0b1111111111111111
ori $t1, $t1, 0b1111111011101011
and $s2, $s2, $t1

# and we OR the new c with $t0
or $s2, $s2, $t0
```

li = lui + ori (to set lower bits)

```
li $8, 0x3BF20

#translates into
lui $at, 0x0003
ori $8, $at, 0xBF20

la $a0, address

#translates into
lui $at, 4097 (0x1001 → upper 16 bits of $at).
ori $a0, $at, disp

move $1, $2
#translates to
add $1, $2, $0
```

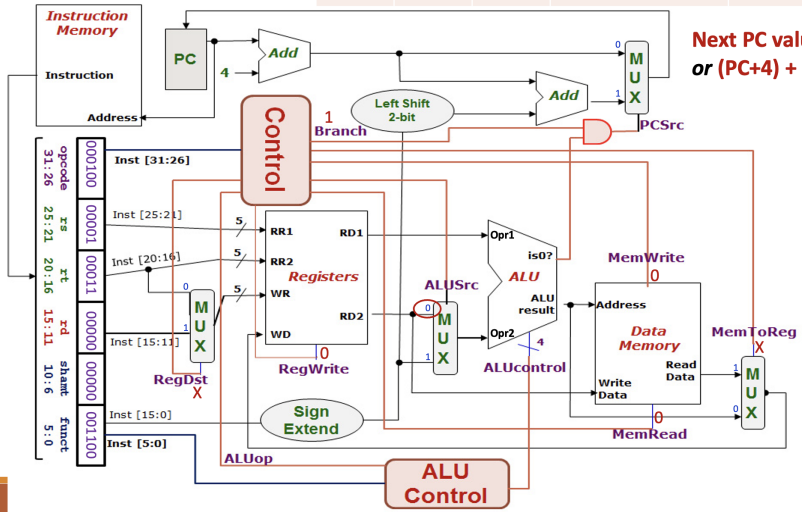
Tutorial 5



`beq $1, $3, 12` | If (R[rs]==R[rt]) PC=PC+4+BrAddr | `000100 00001 00011 0000000000001100`

Q1(ii)

Registers File				ALU		Data Memory	
RR1	RR2	WR	WD	Opr1	Opr2	Address	Write Data
$1	$3	$3 or $0	[$1]-[$3] or MEM([$1]-[$3])	[$1]	[$3]	[$1] – [$3]	[$3]



Next PC value = PC + 4
***or* (PC+4) + (12×4)**

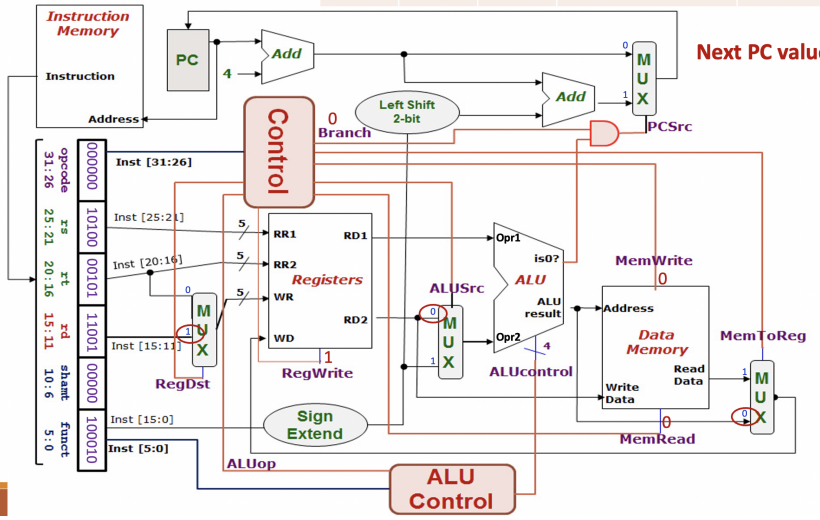
RegDest	X
RegWrite	0
ALUSrc	0
MemRead	0
MemWrite	0
MemToReg	X
Branch	1
ALUop	01
ALUcontrol	0110

6

`sub $25, $20, $5` | R[rd] = R[rs] – R[rt] | `000000 10100 00101 11001 00000 100010`

Q1(iii)

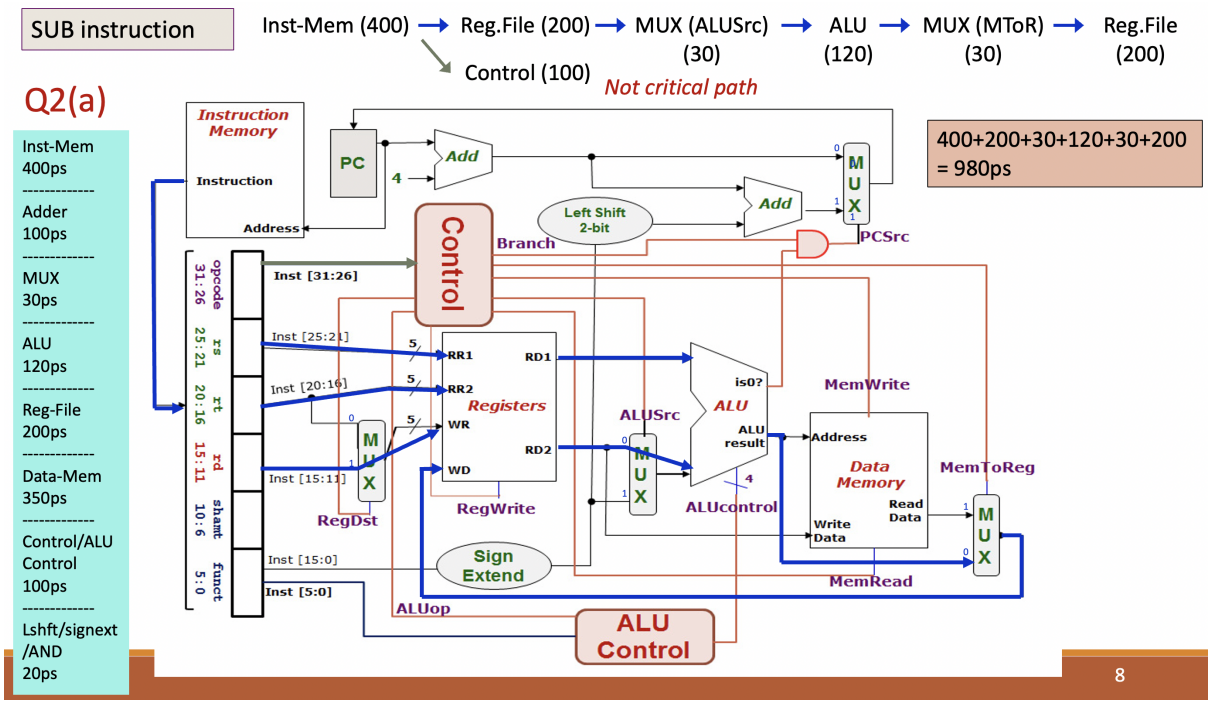
Registers File				ALU		Data Memory	
RR1	RR2	WR	WD	Opr1	Opr2	Address	Write Data
$20	$5	$25	[$20] – [$5]	[$20]	[$5]	[$20] – [$5]	[$5]



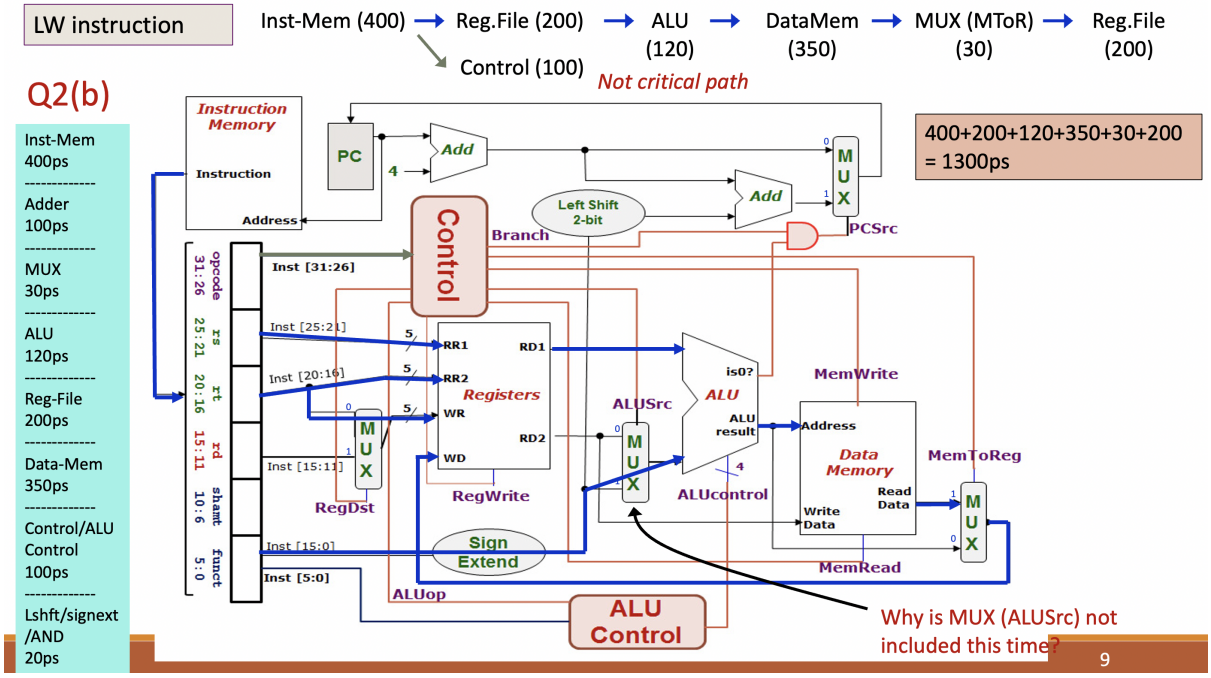
Next PC value = PC + 4

RegDest	1
RegWrite	1
ALUSrc	0
MemRead	0
MemWrite	0
MemToReg	0
Branch	0
ALUop	10
ALUcontrol	0110

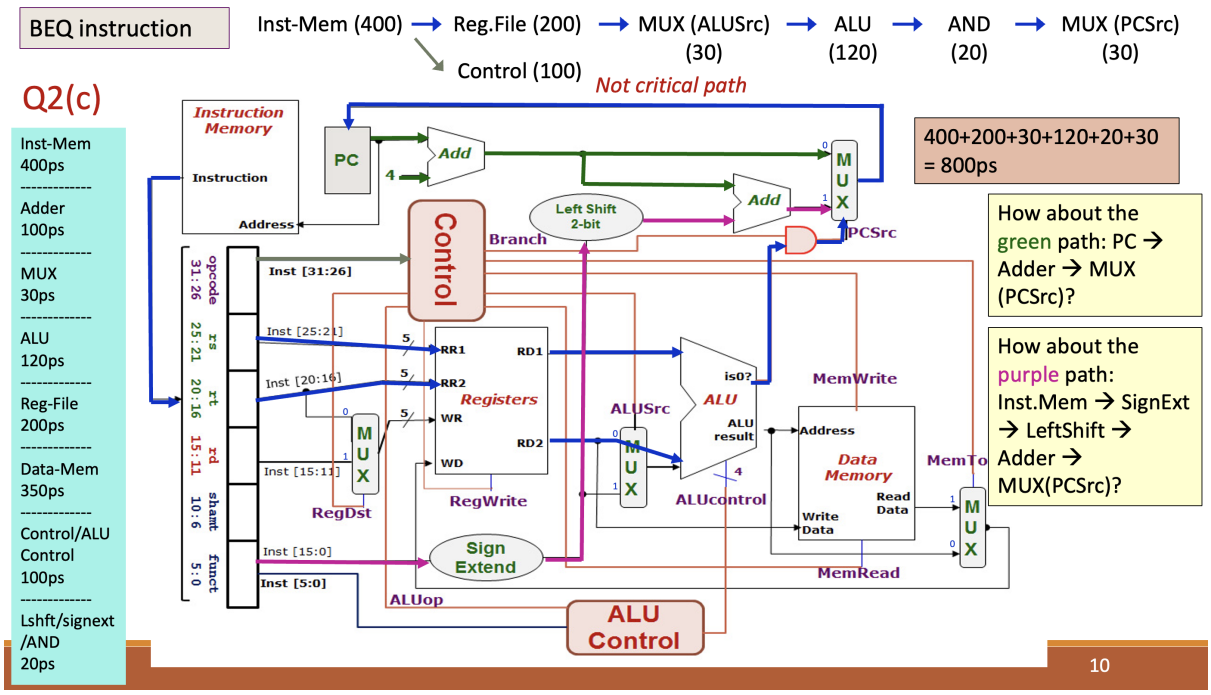
7



MUX (RegDest) is not included as WR is not needed for ALU → not urgent to load WR; when WR is needed at the end, it has already been loaded → no need include the time



MUX(ALUSrc) not included as it is not the critical path (has loaded completely while reg file is been loaded); time taken for sign extend < reg file → not critical



green path is very fast, not critical

20/21 Midterm

▼ We wish to implement a no-operation (NOP) instruction in the MIPS datapath that, when executed, effectively does nothing except waste time (this sounds strange but is actually very useful in applications that must meet strict minimum timing requirements). Choose all the signal combinations that would correctly implement NOP (note: X = Don't care.). NOP is implemented as an R-format instruction with a function code of 0x3F. (Note: Here we only consider the specifications for the signal values; we do not consider actual implementation yet).

1. **+RegDst=1, RegWrite=0, AluSrc=0, MemWrite=0, MemRead=0, MemToReg=0,**
   PCSrc=0
2. RegDst=1, RegWrite=X, AluSrc=X, MemWrite=0, MemRead=0, MemToReg=X,
   PCSrc = 0
3. RegDst=0, RegWrite=0, ALuSrc=1, MemWrite=0, MemRead=0, MemToReg=1,

PcSrc=1

4. RegDst=X, RegWrite=0, AluSrc=X, MemWrite=X, MemRead=0, MemToReg=X,
   PCSrc=0
5. **+RegDst=X, RegWrite=0, AluSrc=X, MemWrite=0, MemRead=0, MemToReg=1,**
   PCSrc=0

Reasoning: NOP works as long as RegWrite, MemWrite are both 0, since nothing
(register nor memory) will be changed.

Although the function code is 0x3F (and hence the Immed value is 0x3F), this will
not add (4 x 0x3F) to PC+4 even if PCSrc = 1, because branch will be 0 since this
is not a branch (beq/bne) instruction.

need both PCSrc and branch to be 1 (AND) to add immediate the PC

In this section we assume a processor with a fixed 16-bit instruction length, 16 registers, and the following 3 instruction classes:

Class A: Two registers.
Class B: One register.
Class C: One register, one 8-bit immediate value.

In all cases we assume that the encoding space for opcodes is fully utilized.

Class A:

8 bits	4 bits	4 bits
Op	Reg1	Reg2

Class B:

12 bits	4 bits
Op	Reg1

Class C:

4 bits	4 bits	8 bits
Op	Reg1	Immed

Most restrictive C->A->B
Least restrictive B->A->C

Minimum number:

- Maximize class C, but set aside 1111 for class A and B.
  - So class C opcodes = 0000 to 1110 = **15** instructions.
- Maximize class A, but set aside 11111111 for class B.
  - So class A opcodes = 11110000 to 11111110 = **15** instructions.
- Maximize class B,
  - so class B opcodes = 111111110000 to 111111111111 = **16** instructions.
- Total = **46** instructions.

Maximum number:

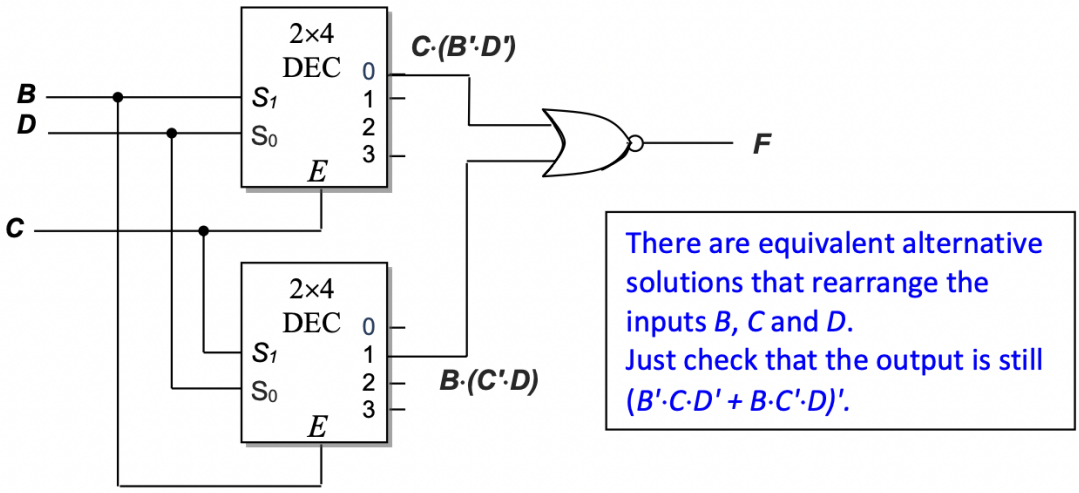
- Start with class B. # of opcodes = 2^12 = **4096**.
- Set aside 1111 xxxx yyyy for class C.
  - This is 2^8 = **256** opcodes.
- Class B ranges from 0000 0000 0000 to 1110 1111 1111.
  - Set aside 1110 1111 zzzz for class A = 2^4 = **16** opcodes.
- Total number of opcodes = 4096 – 256 – 16 + 1 + 1 = **3826** opcodes.

Tutorial 8

Implement the following Boolean function using the fewest number of **2x4 decoder with 1-enable and normal outputs**, and at most two logic gates.

$F(A,B,C,D) = \sum m\ (0,1,3,4,6,7,8,9,11,12,14,15)$

It is easier to think about implementing F' (which is $\Sigma m(2,5,10,13)$ or $b'{\cdot}c{\cdot}d' + b{\cdot}c'{\cdot}d$), and then adding an inverter to invert it back to F.



Tutorial 10

```
     # register $s0 contains a 32-bit value
     # register $s1 contains a non-zero 8-bit value
     # at the right most (least significant) byte
     add  $t0, $s0, $zero     #inst A
     add  $s2, $zero, $zero   #inst B
lp:  bne  $s2, $zero, done    #inst C
     beq  $t0, $zero, done    #inst D
     andi $t1, $t0, 0xFF      #inst E
     bne  $s1, $t1, nt        #inst F
     addi $s2, $s2, 1         #inst G
nt:  srl  $t0, $t0, 8         #inst H
     j    lp                  #inst J
done:
```

Q1a. Data forwarding. No control hazard mechanism.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
add $t0, $s0, $0	F	D	E	M	W																
add $s2, $0, $0		F	D	E	M	W															
lp: bne $s2, $0, done			F	D	E	M	W														
beq $t0, $0, done							F	D	E	M	W										
andi $t1, $t0, 0xFF											F	D	E	M	W						
bne $s1, $t1, nt												F	D	E	M	W					
~~addi $s2, $s2, 1~~	Note that the **addi** instruction is not executed.																				
nt: srl $t0, $t0, 8																F	D	E	M	W	

20 cycles

branch decision made at M stage → no need stall → data forwarded from E to E

Q1d. Data forwarding. No control hazard mechanism.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
add	F	D	E	M	W																
add		F	D	E	M	W															
lp: bne .. done			F	D	E	M	W														
beq .. done							F	D	E	M	W										
andi											F	D	E	M	W						
bne .. nt												F	D	E	M	W					
~~addi~~																					
nt: srl																F	D	E	M	W	
j lp																	F	D	E	M	W
lp: bne .. done																					F

18 cycles

4 iterations.

Total number of cycles = 2 + (4 × 18) + 9 = 83 cycles.

Data forwarding. Branch prediction – predict not taken.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
add $t0, $s0, $0	F	D	E	M	W										
add $s2, $0, $0		F	D	E	M	W									
lp: bne $s2, $0, done			F	D	E	M	W								
beq $t0, $0, done				F	D	E	M	W							
andi $t1, $t0, 0xFF					F	D	E	M	W						
bne $s1, $t1, nt						F	D	E	M	W					
addi $s2, $s2, 1							F	D	E	*	*				
nt: srl $t0, $t0, 8								F	D	*	*	*			
j lp									F	*	*	*	*		
nt: srl $t0, $t0, 8										F	D	E	M	W	

wrong prediction → following instructions get flushed

Q1e. Data forwarding. Branch prediction – predict not taken.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
add $t0, $s0, $0	F	D	E	M	W										
add $s2, $0, $0		F	D	E	M	W									
lp: bne $s2, $0, done			F	D	E	M	W								
beq $t0, $0, done				F	D	E	M	W							
andi $t1, $t0, 0xFF					F	D	E	M	W						
bne $s1, $t1, nt						F	D	E	M	W					
addi $s2, $s2, 1							F	D	E	*	*				
nt: srl $t0, $t0, 8								F	D	*	*	*			
j lp									F	*	*	*	*		
nt: srl $t0, $t0, 8										F	D	E	M	W	
j lp											F	D	E	M	W
lp: bne $s2, $0, done															F

12 cycles

4 iterations.

Total number of cycles = 2 + (4 × 12) + 6 = 56 cycles.

```
// $s0 = base address of array A
// $s1 = base address of array B
// $s2 = n (size of each array)

int i;
for (i=n-1; i>=0; i-=2) {
  if (B[i]%4 == 3)
    A[i] = A[i] + 1;
```

```
  else
    A[i] = A[i] + B[i];
}
```

```
        beq   $s2, $zero, End    # Inst1
        addi  $t8, $s2, -1       # Inst2
        sll   $t8, $t8, 2        # Inst3
Loop:   add   $t0, $s0, $t8      # Inst4
        add   $t1, $s1, $t8      # Inst5
        lw    $t2, 0($t0)        # Inst6
        lw    $t3, 0($t1)        # Inst7
        andi  $t4, $t3, 3        # Inst8
        addi  $t4, $t4, -3       # Inst9
        beq   $t4, $zero, A1     # Inst10
        add   $t2, $t2, $t3      # Inst11
        j     A2                 # Inst12
A1:     addi  $t2, $t2, 1        # Inst13
A2:     sw    $t2, 0($t0)        # Inst14
        addi  $t8, $t8, -8       # Inst15
        slt   $t7, $t8, $zero    # Inst16
        beq   $t7, $zero, Loop   # Inst17
End:
```

Data dependency

Control dependency

Q2c. beq at Inst10 branches to A1

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
I1: beq	F	D	E	M	W																							
I2: addi			F	D	E	M	W																					
I3: sll				F	D	E	M	W																				
I4: Loop: add					F	D	E	M	W																			
I5: add						F	D	E	M	W																		
I6: lw							F	D	E	M	W																	
I7: lw								F	D	E	M	W																
I8: andi									F	D		E	M	W														
I9: addi										F	D		E	M	W													
I10: beq A1											F			D	E	M	W											
I11: add																												
I12: j A2																												
I13: A1: addi															F	D	E	M	W									
I14: A2: sw																F	D	E	M	W								
I15: addi																	F	D	E	M	W							
I16: slt																		F	D	E	M	W						
I17: beq Loop																			F		D	E	M	W				

24 cycles

early branching → stall → E to D