

# CS2100 Finals Cheatsheet 2021/22 Sem 1

## Boolean Algebra

### Logic Circuit

- Advantages of digital circuits over analogue circuit:
  - More reliable (simpler circuits, less noise)
  - Specified accuracy (determinable)
  - Abstraction can be applied using boolean algebra
  - Easy design, analysis and simplification of digital circuit
- Combinational:** No memory, output depends solely on input (e.g. gates, decoders, multiplexers, adders and multipliers)
- Sequential:** with memory, output depends on both input and current state (e.g. counters, registers, memories)

### Precedence of operators

- Precedence from highest to lowest
  - Not ( $'$ )
  - And ( $\cdot$ )
  - Or ( $+$ )
- Use parenthesis to overwrite precedence

### Laws

- Identity:**  $A + 0 = A$  and  $A \cdot 1 = A$
- Inverse/Complement:**  $A + A' = 1$  and  $A \cdot A' = 0$
- Commutative:**  $A + B = B + A$  and  $A \cdot B = B \cdot A$
- Associative:**  $A + (B + C) = (A + B) + C$  and  $A \cdot (B \cdot C) = (A \cdot B) \cdot C$
- Distributive:**  $A + (B \cdot C) = (A + B) \cdot (A + C)$  and  $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$
- Duality** (not a real law): If we flip AND/OR operators and flip the operands (0 and 1), the boolean equation still holds

### Theorems

- Idempotency:  $X + X = X$  and  $X \cdot X = X$
- One/Zero Element:  $X + 1 = 1$  and  $X \cdot 0 = 0$
- Involution:  $(X')' = X$
- Absorption 1:  $X + (X \cdot Y) = X$   
 $X \cdot (X + Y) = X$
- Absorption 2:  $X + (X' \cdot Y) = X + Y$   
 $X \cdot (X' + Y) = X \cdot Y$
- DeMorgan's (can be used on  $> 2$  variables):  
 $(X \cdot Y)' = X' + Y'$   
 $(X + Y)' = X' \cdot Y'$
- Consensus:  
 $(X \cdot Y) + (X' \cdot Z) + (Y \cdot Z) = (X \cdot Y) + (X' \cdot Z)$   
 $(X + Y) \cdot (X' + Z) \cdot (Y + Z) = (X + Y) \cdot (X' + Z)$

### Standard forms

- Literals: Boolean variable on its own or in its complemented form (e.g.  $X, X', Y, Y'$ )
- Product term: A single literal or a product of several literals (e.g.  $X, X \cdot Y \cdot Z', A \cdot B'$ )
- Sum term: A single literal or sum of several literals (e.g.  $X, X + Y + Z', A + B'$ )
- Sum-Of-Products (SOP): Product term or a logical sum of product terms (e.g.  $X, X + (Y \cdot Z'), X \cdot Y' + X' \cdot Y \cdot Z$ )
- minterm: Product term that contains  $n$  literals from all the variables (e.g. 2 variable  $X$  and  $Y$ , the minterms are  $X' \cdot Y', X' \cdot Y$ )

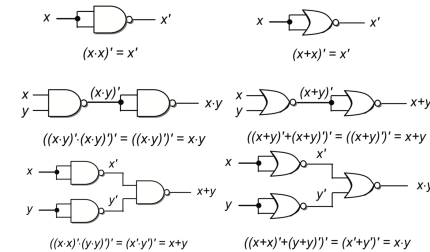
- Product-Of-Sum (POS): Sum term or a logical product of sum terms (e.g.  $X, X \cdot (Y + Z'), (X + Y') \cdot (X' + Y + Z)$ )
- Maxterm: Sum term that contains  $n$  literals from all the variables (e.g. On 2 variable  $X$  and  $Y$ , the maxterms are  $X' + Y', X' + Y, X + Y'$ )
- $Mx = mx'$  because of De Morgan's
- In general, with  $n$  variables we have up to  $2^n$  minterms and  $2^n$  maxterms
- Sum of 2 distinct Maxterms is 1 e.g.  $M1234 + M1120 = 1$
- Product of 2 distinct minterms is 0 e.g.  $m1234 \cdot m1120 = 0$

x	y	Minterms		Maxterms	
		Term	Notation	Term	Notation
0	0	$x' \cdot y'$	m0	$x + y$	M0
0	1	$x' \cdot y$	m1	$x + y'$	M1
1	0	$x \cdot y'$	m2	$x' + y$	M2
1	1	$x \cdot y$	m3	$x' + y'$	M3

- NOTE:**  $XOR = A' \cdot B + A \cdot B'$  OR  $A + B \cdot A' + B'$

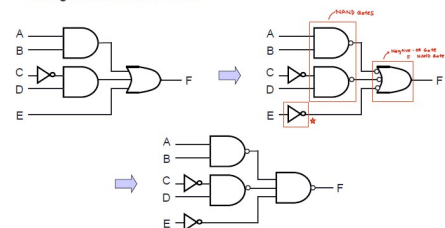
## Logic Circuits

### Gates



- AND, OR, NOT is a complete set of logic
- NAND is a complete set of logic
- NOR is a complete set of logic
- SOP can be implemented with 2-level AND – OR circuit or 2-level NAND circuit

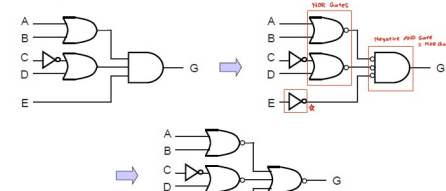
- Example:  $F = A \cdot B + C' \cdot D + E$
- Using 2-level NAND circuit



- POS can be implemented with 2-level OR – AND circuit or 2-level NOR circuit

- Example:  $G = (A+B) \cdot (C'+D) \cdot E$

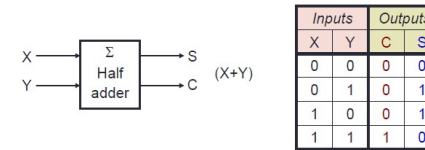
- Using 2-level NOR circuit



- With negated outputs, use NAND to simulate OR and NOR to simulate AND

## Simplification

### Half Adder



- $C = X \cdot Y$
- $S = X' \cdot Y + X \cdot Y' = X \oplus Y$

### Gray Code

- Unweighted (not an arithmetic code)
- Only a single bit change from one code value to the next
- Not restricted to decimal digits:  $n$  bits  $\rightarrow 2^n$  values
- Good for error detection

- Example: 4-bit standard Gray code

Decimal	Binary	Gray Code	Decimal	Binary	Gray code
0	0000	0000	8	1000	1100
1	0001	0001	9	1001	1101
2	0010	0011	10	1010	1111
3	0011	0010	11	1011	1110
4	0100	0110	12	1100	1010
5	0101	0111	13	1101	1011
6	0110	0101	14	1110	1001
7	0111	0100	15	1111	1000

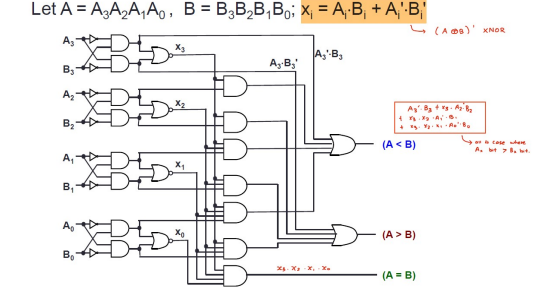
### K-Maps

- Prime Implicant (PI)** is a product term formed by combining the *maximum* possible no. of minterms (largest group)
- Essential Prime Implicant (EPI)** is a PI that includes at least one minterm not covered by any other group
- Grouping  $2^n$  cells (group must have size in powers of 2) eliminates  $n$  variables
- Algorithm to find SOP Expressions
  - Circle all prime implicants on the K-map
  - Identify and select all essential prime implicants for the cover
  - Select a minimum subset of remaining prime implicants to complete the cover
- EPIs are counted only by checking 1s, **not** Xs

## Combinatorial Circuits

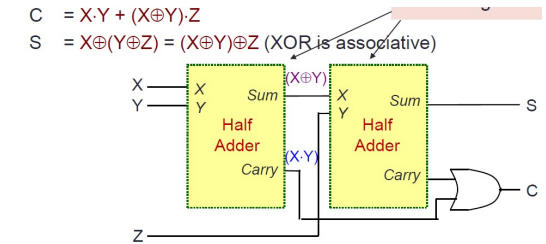
### Comparator

compares 2 unsigned values  $A$  and  $B$ , to check if  $A_i B$ ,  $A=B$ , or  $A_i B$ .



### Full Adder

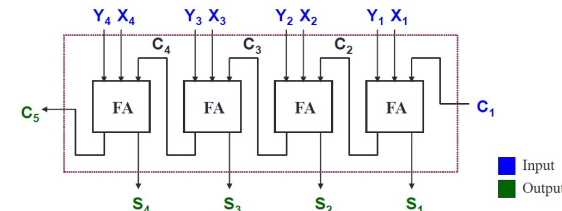
Full adder takes in 3 input ( $X, Y$  and Carry in) and returns the number of 1s in the inputs.



Full Adder made from two Half-Adders (+ an OR gate).

### Parallel Adder

Parallel adder adds two 4-bit numbers together and a carry-in, to produce a 5-bit result.



- Created by cascading 4 full adders
- Carry is propagated by cascading carry from one full adder to the next

**Delays:** Given a logic gate with delay  $t$ , and inputs are stable at times  $t_1, t_2, \dots, t_n$ , then earliest time where output is stable =  $\max(t_1, t_2, \dots, t_n) + t$

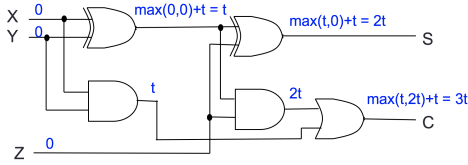


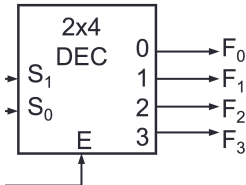
Figure 1: Delay of full adder circuit

In general, an  $n$ -bit ripple carry parallel adder will experience delay of:  $S_n = (2(n-1) + 2)t, C_{n+1} = (2(n-1) + 3)t$  and a maximum delay of  $(2(n-1) + 3)t$

## MSI Components

### Decoder

- Convert binary information from  $n$  input lines to (a maximum of)  $2^n$  output lines
- Generates  $2^n$  minterms of  $n$  input variables
- Can be used to implement functions by generating minterms using **active high** decoder and using an **OR** gate to form a sum-of-minterm, or a **NOR** gate to form a product-of-maxterms function



**Note:** When enable is 0 for **one-enable** decoder, all output will be 0.

$F_0$	$F_1$	$F_2$	$F_3$	$C_1$	$C_0$
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1

$$C_1 = F_2 + F_3$$

$$C_0 = F_1 + F_3$$

### Encoder

- Encoding is the converse of decoding → Given a set of input lines, of which only one is high, encoder provides a code that corresponds to input line
- Contains  $2^n$  input lines and  $n$  output lines and is implemented with **OR** gates

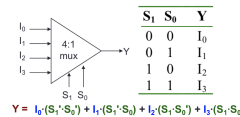
### Priority Encoder

**Note:** If all inputs are 0, the input combination is considered **invalid**

Inputs				Outputs		
D <sub>0</sub>	D <sub>1</sub>	D <sub>2</sub>	D <sub>3</sub>	f	g	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

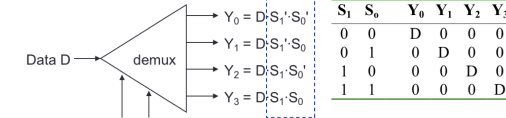
### Multiplexer

Use minterm as selection line, using 0/1 as inputs. For smaller size multiplexer, use one of the variables for input lines.



### Demultiplexer

- Demultiplexers are identical to decoder with enable



## Sequential Logic

### Two types of sequential circuits

- Synchronous:** Outputs change only at specific time
- Asynchronous:** Outputs change at any time

### Self-correcting circuit

From any unused state, a circuit is able to transition to a valid (used) state after finite number of transitions

### S-R Latch

- Asynchronous device
- When  $Q = \text{HIGH}$ , latch is in **SET** state,  $Q = \text{LOW}$ , latch is in **RESET** state
- Drawback:** Invalid condition exists and must be avoided before circuit is stable
- $Q(t+1) = S + R' \cdot Q$ ,  $S \cdot R = 0$

S	R	$Q^+$	
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	indeterminate	

Characteristic table of S-R Latch

### D Latch

- Asynchronous device
- Similar to S-R latch but make input of R equal to S'
- Eliminates the undesirable condition of invalid state in S-R latch
- $D = \text{HIGH} \rightarrow$  latch is SET,  $D = \text{LOW} \rightarrow$  latch is RESET
- Q "follows" D input

EN	D	$Q^+$	
1	0	0	Reset
1	1	1	Set
0	X	$Q(t)$	No change

Characteristic table of D Latch

### Flip-Flops

- Synchronous bi-stable devices
- Change state at **rising edge** or **falling edge** of clock signal
- For  $m$  flip-flops, up to  $2^m$  states exist.
- SR has invalid code while JK uses that for the toggle code
- Negative input for  $Clock \rightarrow$  flip-flop is negative edge-triggered
- J-K Flip-flop:  $Q^+ = J \cdot Q' + K' \cdot Q$
- T Flip-flop:  $Q^+ = T \cdot Q' + T' \cdot Q$

## Design and analysis of Sequential Logic

- Analysis:** Start from circuit diagram  $\rightarrow$  derive state table/state diagram
- Design:** Start from a set of specifications (in the form of state equations, table or diagrams)  $\rightarrow$  derive logic circuit
- Characteristic tables** are use in analysis
- Excitation tables** are use in analysis
- Number of flip-flops needed for  $m$  states =  $\lceil \log_2(m) \rceil$
- Number of input/output:
  - Only 1 number tagged to each transition arrow in state diagram  $\rightarrow$  1 input
  - State diagram shows  $x/y$  tagged to each transition arrow  $\rightarrow$  1 input, 1 output

### Characteristic table

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
1	1

T	$Q(t+1)$
0	$Q(t)$
1	$Q(t)'$

### Excitation Tables

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

(a) S-R flip-flop.

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

(b) J-K flip-flop.

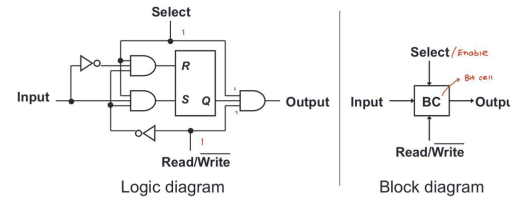
$Q$	$Q^+$	D
0	0	0
0	1	1
1	0	0
1	1	1

(c) D flip-flop.

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

(d) T flip-flop.

### RAM



- Static RAM use flip-flops as memory cells
- Dynamic RAM uses capacitor charges to represent data, simple in circuitry but have to be constantly refreshed
- For BC, Write is 0, Read is 1
- 1K\*8 RAM  $\Rightarrow$  1024 words\*8bits
- In 12 bit address to 4K\*8 RAM constructed using 1K\*8 blocks, the 2 most significant bits are fed into decoder to determine which block to use.

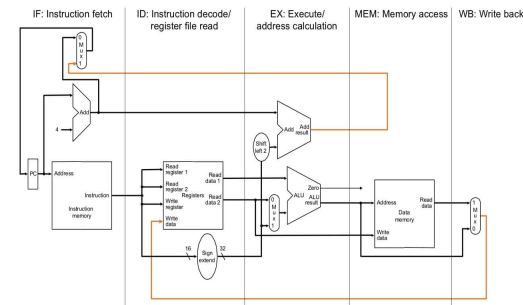
## Pipelining

### Introduction

- Pipelining does not help with latency of single task but it helps with the throughput of entire workout (same amount of time, but more output)
- Multiple task operating simultaneously using different resources
- Possible delays:
  - Pipeline rate limited by slowest pipeline stage
  - Stalls for dependencies

### MIPS Pipeline stages

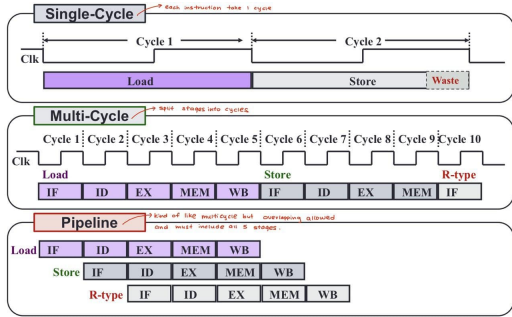
- Five Execution Stages:
  - IF: Instruction Fetch
  - ID: Instruction Decode and Register Read
  - EX: Execute
  - MEM: Access operand in data memory
  - WB: Write back to register
- Each execution stage take 1 clock cycle



### Pipeline Datapath

- Additional registers in datapath used by the same instruction between 2 stages in pipeline
- Each pipeline register is  $>32$ bits
- IF Stage:**  $IF/ID$  stores Instruction read from InstrMemory[PC] &  $PC + 4$
- ID stage:**
  - Beginning of cycle:  $IF/ID$  supplies register numbers for reading 2 registers & 16bit offset to be sign-extended to 32bits
  - End of cycle:  $ID/EX$  stores data values read from register file & 32bit immediate &  $PC + 4$
- EX Stage:**
  - Beginning of cycle:  $ID/EX$  supplies data values read from register file & 32bit  $Imm$  &  $PC + 4$
  - End of cycle:  $EX/IM$  stores  $(PC+4) + (Imm \times 4)$  & ALU result &  $isZero?$  signal & Data read 2 from register file
- MEM Stage:**
  - Beginning of cycle:  $EX/MEM$  supplies  $(PC+4) + (Imm \times 4)$  & ALU result &  $isZero?$  signal & Data read 2 from register file
  - End of cycle:  $MEM/WB$  stores ALU result & Memory read data
- WB Stage:**
  - Beginning of cycle:  $MEM/WB$  supplies ALU result & Memory read data
  - End of cycle: Result is written back to register
  - Note:** Write register number is passed through the entire pipeline from  $ID/EX$  stage until its needed in  $WB$  stage

## Different Implementations



## Performance

- If cycle/clock time is given, just use that
- Single cycle:**  
Cycle time:  $CT_{seq} = \max(\sum_{k=1}^N T_k)$   
Execution time for  $I$  instructions:  
 $Time_{seq} = I \times CT_{seq}$
- Multi-cycle** [1 stage per cycle, cycle time chosen to be time for longest stage]  
 $CT_{multi} = \max(T_k)$   
 $Time_{multi} = I \times Average\ CPI \times CT_{multi}$
- Pipeline** [Several stages per cycle]  
 $CT_{pipeline} = \max(T_k) + T_d$  where  $T_d$  is the pipeline register overhead  
Cycles needed for  $I$  instructions:  $I + N - 1$   
 $Time_{pipeline} = (I + N - 1) \times CT_{pipeline}$
- If  $N_{instructions} \gg N_{stages}$ ,  
 $Speedup_{pipeline} = \frac{Time_{seq}}{Time_{pipeline}} = N$  (where  $N$  is the number of pipeline stages)

## Hazards and resolution

### Structural Hazard:

- Problem:** Single memory module will cause a clash whereby 2 instructions are accessing the memory at the same cycle.  
**Solution:** Split memory into Data and Instruction memory
- Problem:** Register file is accessed by 2 instructions in the same cycle  
**Solution:** Split cycle into half  $\rightarrow$  Write in first half, read in second half (**Order of writing/reading is important!!**)

### Data dependencies:

- Problem:** A later instruction reading from destination register written by an earlier instruction
- Solution: Forwarding**
- Without data forwarding: If dependent instruction is
  - right before: 2 cycle delay
  - 2 cycles before & instruction in between not dependent: 1 cycle delay
- With data forwarding: If dependent cycle is
  - dependent on 1w: 1 cycle delay
  - otherwise: no delay

## Control dependency:

- Problem:** Incorrect execution of instructions after a branch instruction
- Solution:**
  - Early branching (Make decision in *ID* instead of *MEM* stage)
  - Branch prediction (Assume branch **not** taken)
  - Delayed branching (Slot in instruction with no dependencies)
- Without control measures: 3 cycle delay
- With early branching:
  - 1 cycle delay after branch instruction
  - With forwarding & dependent on non-1w: +1 cycle delay at branch instruction
  - With forwarding & dependent on 1w: +2 cycle delay at branch instruction
  - Without forwarding & dependent on prev instr: +2 cycles of delay at branch instruction
- With branch prediction:
  - 3 cycles delay if no early branching & predict wrongly
  - 1 cycle delay if there is early branching & predict wrongly
- With delayed branch: If  $\exists$  instruction before branch that can be moved into delayed slot, move it. Else, stall/no-op

## Cache

### Register vs SRAM vs DRAM vs Hard Disk



	Capacity	Latency	Cost/GB
Register	100s Bytes	20 ps	\$\$\$\$
SRAM	100s KB	0.5-5 ns	\$\$\$
DRAM	100s MB	50-70 ns	\$
Hard Disk	100s GB	5-20 ms	Cents
<b>Ideal</b>	<b>1 GB</b>	<b>1 ns</b>	<b>Cheap</b>

### Types of locality

- Temporal locality
  - If an item is reference, it will tend to be reference again soon
- Spatial locality
  - If an item is referenced, nearby items will tend to be referenced soon (e.g. Array access)

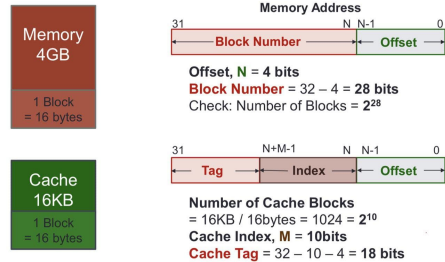
### Terminologies

- Hit:** Data is in cache
  - Hit rate:** Fraction of memory accesses that hit
  - Hit time:** Time to access cache
- Miss:** Data is not in cache
  - Miss rate:** 1 - Hit rate
  - Miss penalty:** Time to replace cache block + hit time
- Hit time < Miss penalty

### Average Access time

$$Rate_{hit} * Time_{hit} + (1 - Rate_{hit}) * Penalty_{miss}$$

## Direct Mapped Cache



### Types of Misses

- Compulsory misses:** On first access to a block; block must be brought into cache
- Conflict misses:** Occur in the case of direct mapped cache or set associative cache, when several blocks are mapped to the same block
- Capacity Misses:** Occur when blocks are discarded from cache as cache cannot contain all blocks needed

### Writing Policy

- Problem:** Cache and main memory are inconsistent when writing to cache
- Solution:**
  - Write-through cache (Slow!)
    - Write to both cache and main memory
    - Problem: Write operates at same speed as main memory
    - Solution: Put write buffer between cache and main memory
  - Write-back cache (slightly better solution)
    - Only write to cache, write to memory when block is replaced
    - Problem: Wasteful to write back to memory every time a cache block is evicted
    - Solution: Add a **Dirty Bit** to each cache block  $\rightarrow$  Write operation changes dirty bit to 1 and only write back to memory when block is replaced & dirty bit is 1

### Handling misses

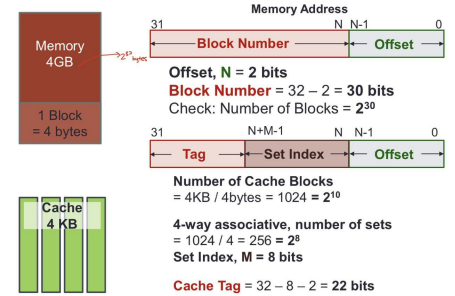
- Read miss: Load data into cache then load from cache to register
- Write Miss
  - Write allocate: Load complete block into cache, change only the required word in cache, write to main memory (using write policy)
  - Write around: Do not load block to cache, write to main memory only

### Block size Trade-off

- Larger block size:
  - + Takes advantage of spatial locality
  - Larger miss penalty: Takes longer time to fill up block
  - If block size too big relative to cache size  $\rightarrow$  too few cache block  $\rightarrow$  miss rate increase

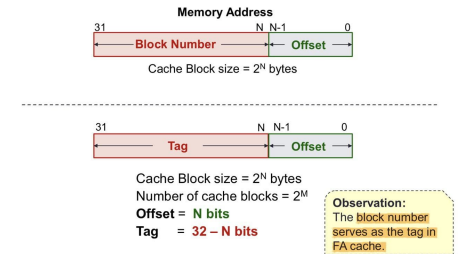
## Set Associative Cache (SAC)

- $N$ -way SAC  $\rightarrow N$  cache blocks per set
- Each memory block maps to a unique cache set
- Within the cache set a memory block can be placed in **any** of the  $N$  cache blocks
- Rule of thumb:** A direct-mapped cache size of  $N$  has about the same miss rate as a 2-way SAC of size  $N/2$



### Fully Associative Cache

- Memory block can be placed in any location in the cache
- No more cache index or cache set index
- Need to search through all cache blocks for memory access
- No conflict misses



### Miss Rates

- Cold/compulsory miss remains the same irrespective of cache size/associativity
- For the same cache size, conflict miss goes down with increasing associativity
- Conflict miss is 0 for FA caches
- For the same cache size, capacity miss remains the same irrespective of associativity.
- Capacity miss decreases with increasing cache size

### Block Replacement

- Least Recently Used: When replacing a block, choose one which has not been accessed for the longest time (due to temporal locality)
- First in First out
- Random Replacement
- Least Frequently Used

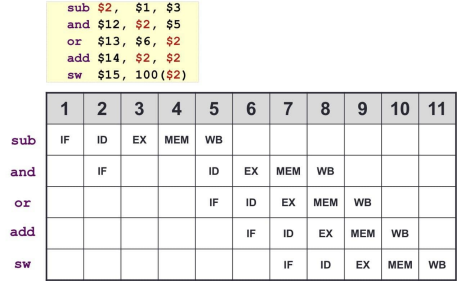
### Overheads in caching

- Overheads are additional administrative information that is stored together with the data
- Usable cache does not account for overheads
- 3 overheads in caching: (1) Valid bit, (2) Tag of memory block, (3) Dirty bit



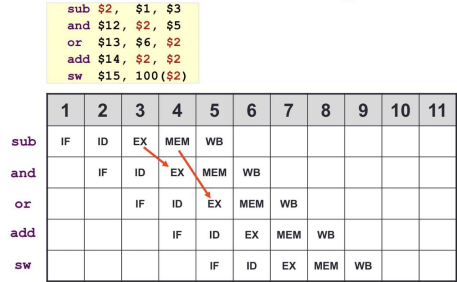
Normal without forwarding

4.3 Exercise #1: Without Forwarding



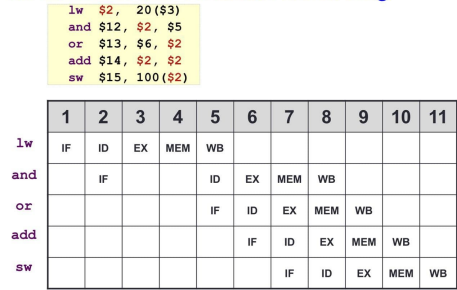
Normal with forwarding

4.3 Exercise #1: With Forwarding



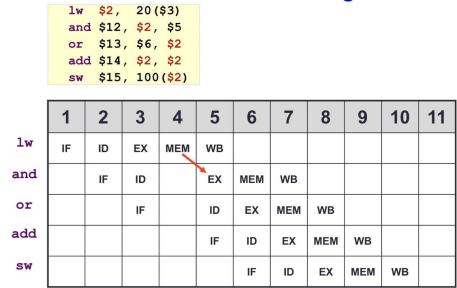
lw without forwarding

4.3 Exercise #2: Without Forwarding

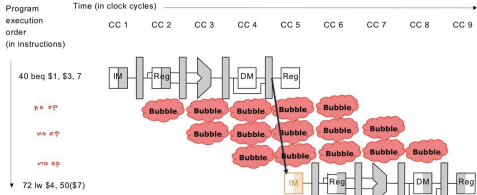


lw with forwarding

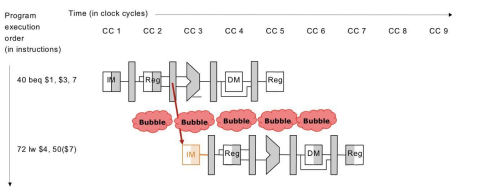
4.3 Exercise #2: With Forwarding



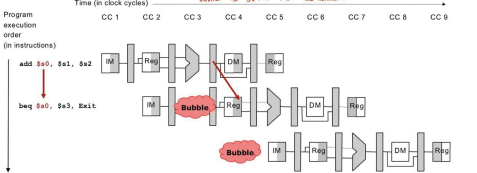
Without control hazard interventions



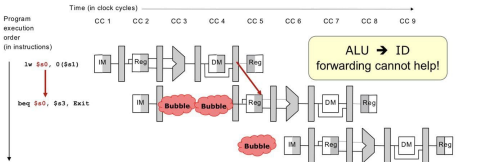
With early branching



Early branch with dependency & forwarding



Early branch with lw dependency & forwarding



4 bit representation

Positive values				Negative values			
Value	Sign-and-Magnitude	1s Comp.	2s Comp.	Value	Sign-and-Magnitude	1s Comp.	2s Comp.
+7	0111	0111	0111	-0	1000	1111	-
+6	0110	0110	0110	-1	1001	1110	1111
+5	0101	0101	0101	-2	1010	1101	1110
+4	0100	0100	0100	-3	1011	1100	1101
+3	0011	0011	0011	-4	1100	1011	1100
+2	0010	0010	0010	-5	1101	1010	1011
+1	0001	0001	0001	-6	1110	1001	1010
+0	0000	0000	0000	-7	1111	1000	1001
				-8	-	-	1000

Pipelining control

	EX Stage				MEM Stage			WB Stage	
	RegDat	ALUSrc	ALUop1	ALUop0	Mem Read	Mem Write	Branch	MemToReg	Reg Write
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

ALU Control Signals

ALUcontrol	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR

Endianness

Input	0x DE AD BE EF
Big-Endian	0: DE, 1: AD ...
Little-Endian	0: EF, 1: BE ...

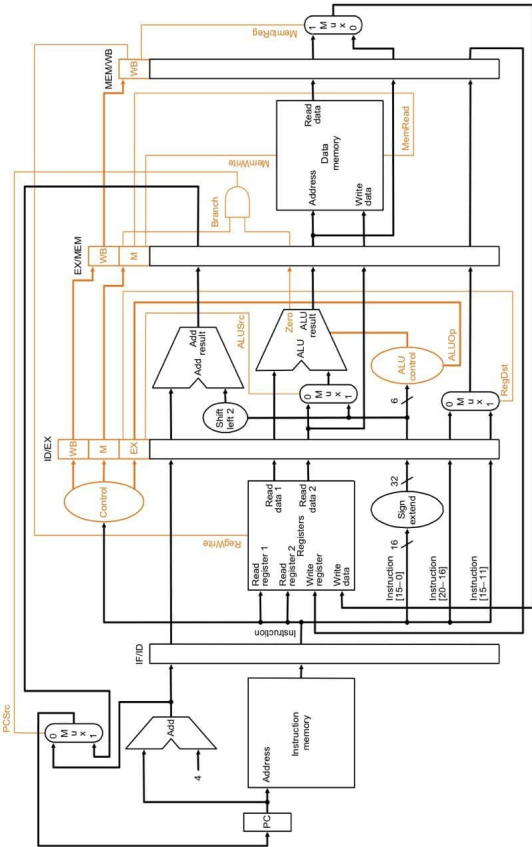


Figure 2: Datapath with pipeline control