CS2100 Computer Organization
Help Sheet for Final Examinations

L01 – Introduction

Layers of Abstraction. A single layer omits unnecessary details and presents a simpler interface for the higher layers.

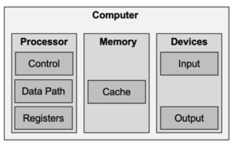


Figure: Components of a Computer

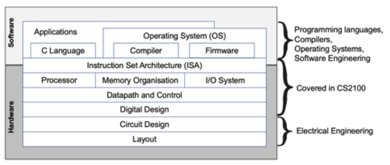


Figure: Layers of Abstraction in Computing

L02 – Overview of C

Uninitialized variables. They have no initial value. Do not assume them to be zero.

Data Type in C		32-Bit Processor	64-Bit Processor
`int`	Size	4 bytes (32 bits)	8 bytes (64 bits)
	Range	-2^{31} to $2^{31}-1$	-2^{63} to $2^{63}-1$
`float`	Size	4 bytes (32 bits)	8 bytes (64 bits)
`double`	Size	8 bytes (64 bits)	16 bytes (128 bits)
`char`	Size	1 byte (8 bits)	
	Range	-2^7 to 2^7. ACSII code has the range from 0 to 255.	

Table: Common Data Types in C

Conversion hierarchy. C defines the hierarchy of `char > int > float > double`. Conversion to the left (aka 'narrowing') may result in some loss of information (eg. 1.23 to 1).

Context	Conversion Rule
For operation `A op B -> C`	If `A` and `B` has the same data type → `C` will have the same data type. For example, `2.0 / 5.0 → 0.4`, `2 / 5 → 0` Or else, the 'lower' data type is promoted to match the 'higher' data type, and `C` will be in the same higher data type. For example, `2.0 / 5 → 0.4` (`5` promoted to `5.0`)
For assignment `A = B`	`B` will be promoted / demoted to match the type of `A`.

Table: Implicit Data Type Conversion Rules in C

L03 – Data Representation and Number Systems

Conversion	Method and Example	
Base b to Decimal	Apply the formal definition for a positional weighted system: $(a_n a_{n-1} \cdots a_0 . f_1 f_2 \cdots f_m)_b$ $= (a_n \times b^n) + (a_{n-1} \times b^{n-1}) + \cdots + (a_0 \times b^0)$ $+ (f_1 \times b^{-1}) + (f_2 \times b^{-2}) + \cdots + (f_m \times b^{-m})$	1011_{16} $= 1 \cdot 16^3$ $+1 \cdot 16^1$ $+1 \cdot 16^0$ $= 4113_{10}$
Decimal to Binary (for integers)	**Repeatedly divide** the quotient by 2 until the quotient is 0. The remainders form the answer: • First remainder as the least significant bit (LSB) • Last remainder as the most significant bit (MSB).	To find 43 in binary, 2 | 43 2 | 21 rem 1 (LSB) 2 | 10 rem 1 2 | 5 rem 0 2 | 2 rem 1 2 | 1 rem 0 | 0 rem 1 (MSB) $(43)_{10} = (101011)_2$
Decimal to Binary (for fractions)	**Repeatedly multiply** the fractional portion of the number by 2, until the fractional product is 0. The carried digits produce the answer: • First carry is the MSB • Last carry is the LSB.	To find 0.3125 in binary, $0.3125 \times 2 = 0.625$ | 0 (MSB) $0.625 \times 2 = 1.25$ | 1 $0.25 \times 2 = 0.50$ | 0 $0.50 \times 2 = 1.00$ | 1 (LSB) $(0.3125)_{10} = (.0101)_2$
Base k to base j	Use decimal system as a bridge, ie. Base k → Base 10 → Base j. Short cuts exist between base 2 ←→ 8, and base 2 ←→ 16 (see below).	

Table: Methods to convert numbers between bases

BASE 10	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Base 16	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Base 2	0	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111

Table: Representations of 0 to 15

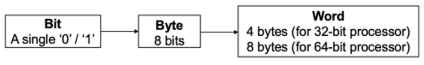

Bit
A single '0' / '1'
Byte
8 bits
Word
4 bytes (for 32-bit processor)
8 bytes (for 64-bit processor)

Figure: Common data storage units in a computer

Representation of values through bits.
N bits can store up to 2^N values. For example, 2 bits → up to 4 values (00, 01, 10, 11).
To represent M values, $\lceil \log_2 M \rceil$ bits are required. For example, 32 values → requires 5 bits.

2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8
1	2	4	8	16	32	64	128	256

Table: Powers of 2

Representation of Integers.

Sign-and-Magnitude Representation

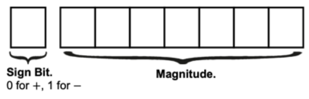


Figure 4: Structure of S&M Representation

For example, in a 1-bit sign and 7-bit magnitude format,
$(00110100)_{sm} = +110100_2$
$(10010011)_{sm} = -10011_2$

Operation	n-bit representation	8-bit representation
Largest	$(01\cdots1)_{sm} = +(2^{n-1}-1)_{10}$	$(01111111)_{sm} = +127_{10}$
Smallest	$(1\cdots1)_{sm} = -(2^{n-1}-1)_{10}$	$(11111111)_{sm} = -127_{10}$
Range	$-(2^{n-1}-1)$ to $2^{n-1}-1$	-127_{10} to $+127_{10}$
Zeroes	$(0\cdots0)_{sm} = +0_{10}$ $(10\cdots0)_{sm} = -0_{10}$	$(00000000)_{sm} = +0_{10}$ $(10000000)_{sm} = -0_{10}$
Negation	Just invert the sign bit.	$(10000101)_{sm} = -5_{10}$ → $(00000101)_{sm} = +5_{10}$ $(00100001)_{sm} = +33_{10}$ → $(10100001)_{sm} = -33_{10}$

Table: Sign-and-Magnitude Representation of Integers

1s-Complement Representation

Operation	n-bit representation	8-bit representation
Positive conversion	For positive binary numbers, their representation in 1s is the same.	$00001100_2 = 00001100_{1s}$
Negative conversion	$-(X)_2 = 2^n - X - 1$ *Note: Calculation is in decimal.*	$-00001100_2 = 2^8 - 12 - 1$ $= 243 = 11110011_{1s}$
Largest	$(01\cdots1)_{1s} = (2^{n-1}-1)_{10}$	$(01111111)_{1s} = +127_{10}$
Smallest	$(10\cdots0)_{1s} = -(2^{n-1}-1)_{10}$	$(10000000)_{1s} = -127_{10}$
Range	$-(2^{n-1}-1)$ to $2^{n-1}-1$	-127_{10} to $+127_{10}$
Zeroes	$(0\cdots0)_{1s} = +0_{10}$, $(1\cdots1)_{1s} = -0_{10}$	$(00000000)_{1s} = +0_{10}$, $(11111111)_{1s} = -0_{10}$
Operation	**n-bit representation**	**4-bit representation**

Addition	Perform binary addition on numbers A and B. If there is a carry out of the MSB, add 1 to the result. Check for overflow.
	
	Overflow occurs if
	• Result is opposite sign of A and B, or
	• Result goes beyond the fixed range of a signed integer.
	Eg. `Positive` add `Positive` → `Negative`, or `Negative` add `Negative` → `Positive`
		`  1100`
		`+ 1000`
		`-----`
		`10100`
		`+    1`
		`-----`
		`  0101`
		
		$(-3)_{10}$
		$+(-7)_{10}$
		$= (-10)_{10}$

Subtraction	Take the 1s-complement of B, and add the 1s-complement of B to A. $A - B = A + (-B)$	
Negation	Just invert every bit.	
Negative?	If most-significant-bit (MSB) is 1, it is a negative value.	

Table: 1s-Complement Representation of Integers

2s-Complement Representation

Operation	n-bit representation	8-bit representation
Positive conversion	For positive binary numbers, their representation in 2s is the same.	$00001100_2 = 00001100_{2s}$
Negative conversion	$-(X)_2 = 2^n - X$ *Note: Calculation is in decimal.*	$-00001100_2 = 2^8 - 12$ $= 244 = 11110100_{2s}$
Largest	$(01\cdots1)_{2s} = +(2^{n-1}-1)_{10}$	$(01111111)_{2s} = +127_{10}$
Smallest	$(10\cdots0)_{2s} = -(2^{n-1})_{10}$	$(10000000)_{2s} = -128_{10}$
Range	-2^{n-1} to $2^{n-1}-1$	-128_{10} to $+127_{10}$
Zeroes	$(0\cdots0)_{2s} = +0_{10}$	$(00000000)_{2s} = +0_{10}$
Addition	Perform binary addition on numbers A and B. Ignore the carry out of the MSB. Check for **overflow**. **Overflow** occurs if • The 'carry in' and the 'carry out' of the MSB are different, or • Result is the opposite sign of A and B.	
Subtraction	Take the 2s-complement of B, and add the 2s-complement of B to A. $A - B = A + (-B)$	
Negation	Invert all the bits, and then add one.	
Negative?	If most-significant-bit (MSB) is 1, it is a negative value.	

Table: 2s-Complement Representation of Integers

Excess Representation. In the Excess-K system, K is known as the bias / offset. To represent a number X in this system, simply add K to X, and then represent the result as binary.
Eg. The 4-bit Excess 8 representation of 7 is $7 + \mathbf{8} = 15 \rightarrow \mathbf{1111}$

IEEE-754 Floating Point Representation.

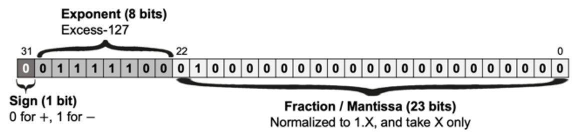


Figure: Structure of IEEE-754 32-Bit Floating Point Representation

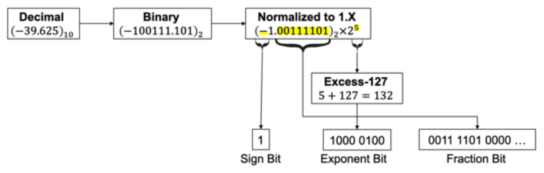
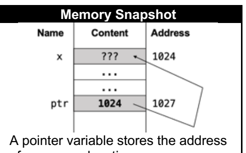


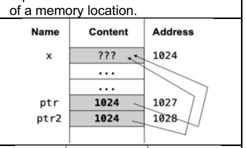
Figure: Obtaining Floating Point Representation of a Decimal Number

Characters. ASCII code was originally defined as a 7-bit sequence → from 0 to 127, can represent 128 characters. It was subsequently extended to 8-bit, and the extended range 128 to 255 can have platform-dependent encoding.

L04 – Function and Pointer

Code	Explanation	Memory Snapshot

`int x;`
`int *ptr;`
`ptr = &x;`
	Declaration of Pointers. `ptr` is an `int` pointer variable that points to variable `x`.
	
	Address-of Operator. The '`&`' operator gives the address of the variable.
		 Name
		Content
		Address
		x
		???
		1024
		ptr
		1024
		1027
		A pointer variable stores the address of a memory location.

`int *ptr2;`
`ptr2 = ptr;`
	Manipulation of Pointers. When `ptr2 = ptr` is executed, content of `ptr` is copied over to `ptr2`.	 Name
		Content
		Address
		x
		???
		1024
		ptr
		1024
		1027
		ptr2
		1024
		1028

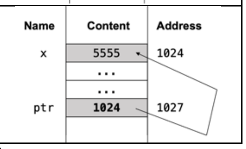
`*ptr = 1234;`
`*ptr = *ptr`
`+ 4321;`
	Dereferencing. With the '`*`' operator, we follow the address stored in a pointer variable and manipulate the destination.	 Name
		Content
		Address
		x
		5555
		1024
		ptr
		1024
		1027

Table: Pointers in C

Pass-By-Value	Pass-By-Address
Simple data types (`int`, `float`, `char`) and Structures are passed-by-value. Cannot change the actual parameter.	Arrays are passed-by-address. Requires the caller to pass in the address of variables using "`&`". Requires dereferencing of pointers in the function.

Table: Parameter Passing into Functions

L05 – Arrays

Code	Explanation

`int myArray[3];`
`int myArray[3] = {1,2,3};`
`int myArray[3] = {1};`
 	
`int myArray[3] = {0};`
	Random values for all elements in array.
	Initialize array during declaration using an initialization list.
	If initialization list shorter than array size, rest of values is zero.
	Use `{0}` to initialize all array items to zero.

`int ia[3] = {3,5,7};`
`int *ptr;`
`ptr = ia;`
 	
`ptr[1] = 333;`
`ptr = &(ia[1]);`
`ptr[1] = 4444;`
	The array name by itself is same as the address of the 0th element.
	Manipulation of an element at index 1 in the array.

`void printLessThan(`
`int a[], int size, int criteria)`
`{…}`
`printLessThan(array, 5, 15);`
	To pass an array as an argument into a function, use the name of the array. In effect, the address of `array[0]` is passed into the function.

Table: Arrays in C

Strings. We use a character array to store multiple characters and add a special terminator character '`/0`' at the end.

Code	Explanation

`char a[6] =`
`{'H','e','l','l','o','!'};`
`char b[7] = "Hello!"`
	String constants are surrounded by a double quote. Remember to count the terminator when declaring size of the string.

Table: Strings in C

L06 – Structure

Code	Memory Model

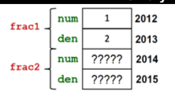
`struct Fraction {`
`    int num;`
`    int den;`
`}`
`struct Fraction frac1 = {1,2};`
`struct Fraction frac2;`
	 frac1
	frac2
	num
	den
	num
	den
	1
	2
	?????
	?????
	2012
	2013
	2014
	2015 Each structure variable has an independent set of the fields. The fields of a structure are placed in **adjacent locations** in memory.

Table: Declaration of Structures in C

Code	Memory Model

`void printFrac(struct Fraction inFrac) {…}`
`printFraction( myFraction );`
	Pass-by-Value. A copy of the actual argument will be made.

`void readFrac(struct Fraction *fptr) {`
`    int n, d;`
`    scanf("%d%d", &n, &d);`
`    fptr->num = n; // (*fptr).num = n;`
`    fptr->den = d; // (*fptr).den = d;`
`}`
`readFraction(&myF);`
	Pass-by-Address. To allow function to modify the actual argument, use the **indirect field selector** '->'.

Table: Structure Passed As A Parameter into a Function

L07: MIPS - Introduction

While Loop	If Loop

`while (condition) {`
`    …;`
`}`
	`Loop:`
	`    if (!condition) branch to Exit;`
	`    do stuff;`
	`    j Loop;`
	`Exit:`

In the MIPS assembly language, there are **32** registers, each 32-bit (4-byte) long. Each word contains 32 bits (4 bytes). Hence, memory registers are 32-bit long.

Several Operations and their Uses	
`addi $s0, $s0, 4`	**Increment / Decrement Operations:** Constant ranges from $[-2^{15}$ to $2^{15}-1]$ and is in **2s-complement**.
`add $s0, $s1, $zero`	**Assignment Operations:** Equivalent to f = g; and the pseudo-instruction move `$s0, $s1`
`sll $t2, $s0, 4`	Equivalent to multiplying by 2^n.
`srl $t2, $s0, 4`	Equivalent to getting the quotient when you divide by 2^n. Constant ranges from $[0$ to $2^5-1]$.
`and $s0, $t1, $t2`	**Masking Operations using Bitwise AND:** Place 0s on the places to be ignored → bits will turn into 0s. Place 1s for interested positions → bits will remain the same as original.
`or $t0, $t1, $t2`	Can be used to force certain bits to 1s.
`nor $t0, $t1, $t2`	Can make the **NOT** operation by `NOR`-ing with $zero.
`xor $t0, $t1, $t2`	Can make the **NOT** operation by `XOR`-ing with all 1s.

`lui $t0, 0xAAAA`
`ori $t0, $t0,`
`0xF0F0`
	Large Constants: Use "load upper immediate" to set the upper 16-bits. Note that `lui` also clears the lower 16 bits. Use "or immediate" to set the lower-order bits.

L08: MIPS – Introduction

Each location in the main memory has an address. Using distinct memory address, we can access a single byte (**byte addressable**), or a single word (**word addressable**). MIPS uses byte addresses, so consecutive word addresses differ by 4.

Word Alignment. Words are aligned in memory if they begin at a byte address that is a multiple of the number of bytes in a word. For example, if a word consists of 4 bytes, then aligned words begin from byte 0, 4, 8, …

Aspect of Main Memory	Value
Size of k-bit address	2^k
Number of memory words in main memory	2^{30}
Length of each MIPS Instruction	32 bits

L09: MIPS – Encoding

R-format (`op $r1, $r2, $r3`)		
Case 1: For `add`, `sub`, `and`, `slt`, etc. the format is `add $rd, $rs, $rt`. **Case 2:** For `srl`, `sll`, etc. the format is `sll $rd, $rt, shamt` and the `rs` field is left as zero.		
`opcode`	6	Partially specifies the instruction, equals to 0 for all R-format instructions.
`rs`	5	**Source Register:** Specify register containing the first operand.
`rt`	5	**Target Register:** Specify register containing the second operand.
`rd`	5	**Destination Register:** Specify register which will receive result of computation.
`shamt`	5	**Shift Amount:** Amount a shift instruction will shift by. Set to zero if no shifting.
`funct`	6	Combined with opcode, will exactly specify the instruction.

I-format (`op $r1, $r2, Immd`)		
Case 1: For instructions involving signed integers or just 16-bit patterns as the Immediate, the format is `addi $rt, $rs, Immd`. **Case 2:** For memory instructions, the format is `lw $rt, Immd($rs)`.		
`opcode`	6	No `funct` field, so `opcode` uniquely specifies an instruction.
`rs`	5	**Source Register:** Specifies the source register operand (if any).
`rt`	5	**Target Register:** Specifies the register to receive result.
`Immd`	16	**Immediate Value:** 16 bits → can represent a constant of up to

The **Program Counter (PC)** is a special register that keeps address of instruction being executed in the processor. Instructions are word-aligned, and the `Immediate` value in branch instructions is interpreted as the number of words. We can branch to $\pm 2^{15}$ words from the PC.

$$PC = \begin{cases} PC + 4, if\ the\ branch\ is\ not\ taken \\ (PC + 4) + (Immediate \times 4), if\ the\ branch\ is\ not\ taken \end{cases}$$

PC-Relative Addressing
For branch instructions, the format is `beq $rs, $rt, Label`. The `Immd` value is equal to the number of instructions to add to (or subtract from) the PC, starting at the instruction following the branch.

J-format (`op, Immd`)		
`opcode`	6	The `opcode` for the `j` instruction.
`target address`	26	We can only specify 26 bits of a 32-bit address.
From the 26-bit Immediate value, we can find the actual 32-bit target address to jump to: $\underbrace{1010}_{\text{Most significant 4bits of PC}}\ \underbrace{00001111000011110000111100}_{\text{26bits Target address specified in instruction}}\ \underbrace{00}_{\text{Default 2bit "00" for word address}}$		Max jump range: 0x0FFFFFFC

L10 - Processor: Datapath

5-Stage MIPS Instruction Execution Cycle			
	`add $3, $1, $2`	`lw $3, 20($1)`	`beq $1, $2, label`
Instruction **F**etch	Use the PC to fetch the instruction from memory. Then, increment the PC by 4 to get the address of the next instruction.		
Decode & Operand Fetch	Read `[$1]` as `opr1` Read `[$2]` as `opr2`	Read `[$1]` as `opr1` Use `20` as `opr2`	Read `[$1]` as `opr1` Read `[$2]` as `opr2`

Execute	`Result`
	`= opr1 + opr2`
		`MemAddr`
		`= opr1 + opr2`
			`Taken = (opr1 == opr2)?`
			`Target = PC + Label*`

Memory Access		Use `MemAddr` to read from memory	
Result **W**rite	`Result` stored in `$3`	Memory data stored in $3	`if Taken: PC = Target`

Table: Stages of Execution for a MIPS Instruction

Clock Signal
Processors utilize a stable clock signal (square wave) for instruction execution. **Cycle Time / Clock Period:** Duration between two consecutive rising edges, measured in seconds **Clock Rate / Clock Frequency:** $\frac{1}{cycleTime} = \frac{\#cycles}{second}$, measured in Hz (1 Hz = $\frac{1\ cycle}{second}$) The PC is read during the first half of the clock period, and the next rising clock edge

L11- Processor: Control

Signal	False (0)	True (1)
`RegDst`	Write register = `Inst[20:16]`	Write register = `Inst[15:11]`
`RegWrite`	No register write	New value will be written
`ALUSrc`	opr2 = Register Read Data 2	opr2 = `SignExt(Inst[15:0])`
`MemRead`	Not performing memory read access	Read memory using ***Address***
`MemWrite`	Not performing memory write operation	`memory[Address]` ← Register Read Data 2
`MemToReg`	Register write data = ALU Result	Register write data = Memory read data
`PCSrc`	Next PC = `PC + 4`	Next PC = `SignExt(Inst[15:0])<<2 + (PC + 4)`

Table: Meanings for the Control Signals

	EX				MEM			WB	
	Reg Dst	ALU Src	AlUop op1	op0	Mem Read	Mem Write	Branch	MemTo Reg	Reg Write
R	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

Table: Respective Control Signals for Various Instruction Types

Opcode	ALUop	Operation	Funct	ALU action	ALUControl
lw	`00`	load word	XXXXX	add	`0010`
sw	`00`	store word	XXXXX	add	`0010`
beq	`01`	branch equal	XXXXX	subtract	`0110`
R-type	`10`	add	10 0000	add	`0010`
R-type	`10`	subtract	10 0010	subtract	`0110`
R-type	`10`	AND	10 0100	AND	`0000`
R-type	`10`	OR	10 0101	OR	`0001`
R-type	`10`	set on less than	10 1010	set on less than	`0111`

Table: Respective Control Signals for Various Instruction Types

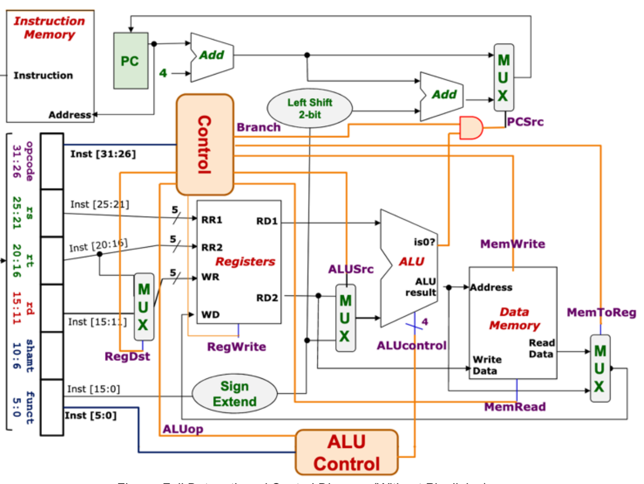


Figure: Full Datapath and Control Diagram (Without Pipelining)

Joshua Chew Jian Xiang

L12 – Pipelining

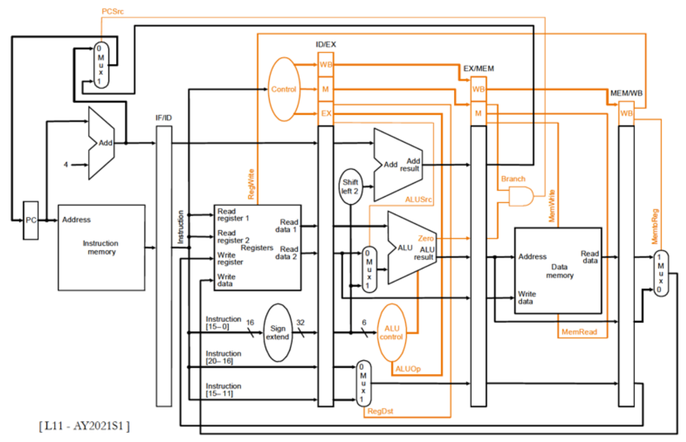


Figure: Datapath and Control with Pipelining

Corrected Datapath. In the original Datapath, the `Write Register` number was originally supplied by the **`IF/ID`** pipeline register. However, it is not the correct write register for the instruction now at the **`WB`** stage. As a solution, we pass the Write Register number from **`ID/EX`** through **`EX/MEM`** to **`MEM/WB`** pipeline register for use in the **`WB`** stage.

Pipeline Control. The control signals are generated in the **`ID`** stage, and are used in the **`EX`**, **`MEM`** and **`WB`** stages. The signals propagate along the pipeline registers until utilized.

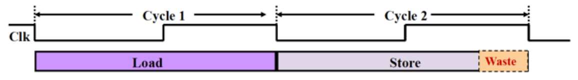
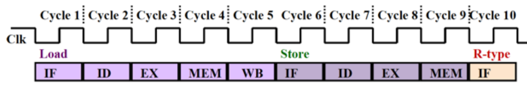
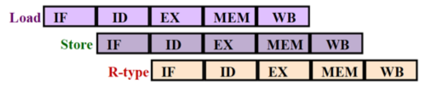
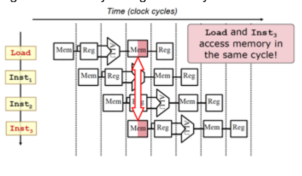
Single Cycle Processor	
 Each instruction takes one cycle to execute.	
Cycle Time	$CT_{seq} = \sum_{k=1}^{N} T_k$ where T_k: Time for operation in stage k, N: Number of stages
Total Execution Time (for **I** instructions)	$Time_{seq} = Cycles \times CycleTime = I \times CT_{seq}$
Multicycle Implementation	
 Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5 Cycle 6 Cycle 7 Cycle 8 Cycle 9 Cycle 10 Clk Load Store R-type IF ID EX MEM WB IF ID EX MEM IF Each execution step takes one clock cycle → Instructions take variable number of cycles for complete execution.	
Cycle Time	$CT_{multi} = \max(T_k)$ which is the longest stage time among N stages.
Total Execution Time (for **I** instructions)	$Time_{multi} = Cycles \times CycleTime = I \times Average\ CPI \times CT_{multi}$ Average CPI refers to average cycle per instruction.
Pipeline Processor	
 Load IF ID EX MEM WB Store IF ID EX MEM WB R-type IF ID EX MEM WB	
Cycle Time	$CT_{pipe} = \max(T_k) + T_d$ where T_d refers to the pipeline overhead.
Cycles Needed (for **I** Instructions)	$I + N - 1$ where $(N-1)$ is the cycles for filling up the pipeline.
Total Execution Time (for **I** instructions)	$Time_{pipe} = Cycle \times CT_{pipe} = (I + N - 1) \times (\max(T_k) + T_d)$
Ideal Speedup A pipeline processor can gain **N** (number of pipeline stages) times speedup.	Assuming that every stage takes the same amount of time ($\sum_{k=1}^{N} T_k = N \times T_k$) and no pipeline overhead ($T_d = 0$), and number of instructions **I** is much larger than the number of stages **N**, $Speedup_{pipe} = \frac{Time_{seq}}{Time_{pipe}} \approx N$

Table: Performance of Various Processor Implementations

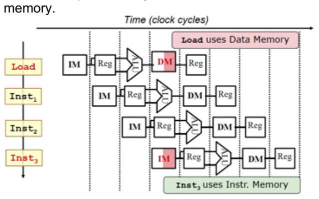
Pipeline Hazards. Problems that prevent the next instruction from immediately following the previous instruction.

Hazard Type 1: Structural Hazard

The simultaneous use of a hardware resource, eg. if there is only a single memory module.



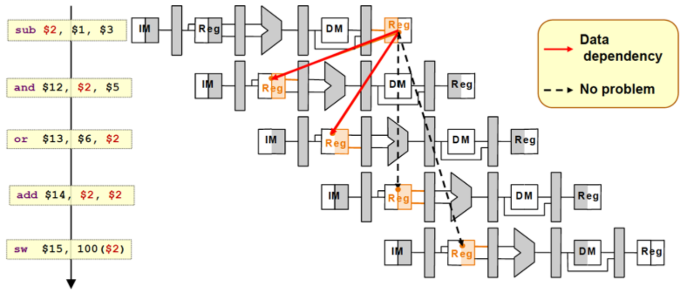
Solution: Split memory into data and instruction memory.



Hazard Type 2: Data Dependency (RAW)

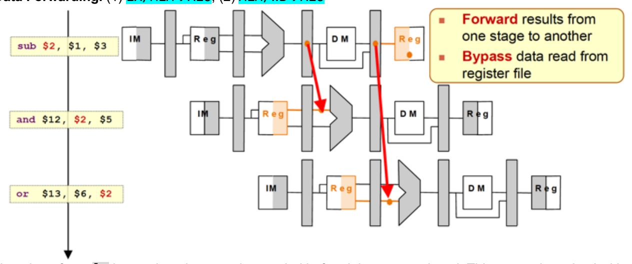
Read-After-Write (RAW) dependency occurs when a later instruction **reads** from the destination register written by an earlier instruction.

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

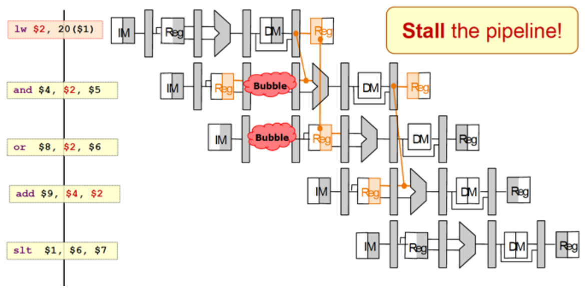


Joshua Chew Jian Xiang

Data Forwarding. (1) EX/MEM→ALU, (2) MEM/WB→ALU


sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
IM
Reg
DM
Forward results from one stage to another
Bypass data read from register file

Note that after a `lw` instruction, data may be needed before it is even produced. This cannot be solved with forwarding (unless you can go back in time).



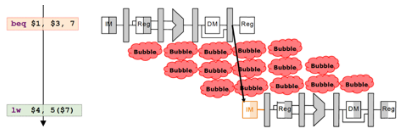
Hazard Type 3: Control Dependency

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

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

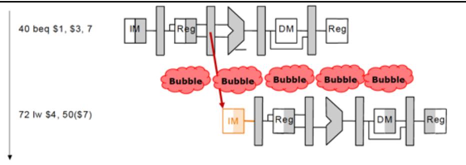
In the original pipeline processor, the decision of which instruction to execute next is made in the **MEM** stage (too late).

Simply stalling the pipeline introduces **3 clock cycles of delay**.

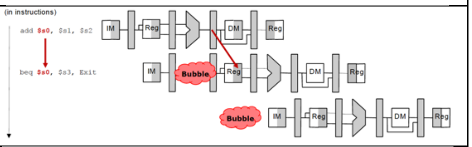


Early Branch Resolution. Move the branch target address calculation to the **ID** stage instead of the **MEM** stage. We also move the register comparison to the **ID** stage (instead of using the ALU).

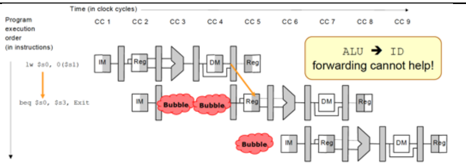
If we wait until the branch decision is known, and then fetch the correct instructions, we are reduced to **one clock cycle delay**.



However, if the register(s) involved in the comparison are produced by the preceding instruction, with **ALU→ID** data forwarding, **one clock cycle delay** still needed.

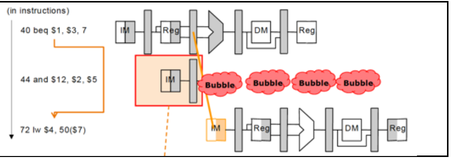


With a **load** followed by a **branch**, and **MEM→ID** data forwarding, we still end up with 3 total stall cycles and no improvement.


Time (in clock cycles)
Program execution order (in instructions)
CC 1 CC 2 CC 3 CC 4 CC 5 CC 6 CC 7 CC 8 CC 9
lw $s0, 0($s1)
beq $s0, $s3, Exit
IM
Reg
DM
Bubble
ALU → ID forwarding cannot help!

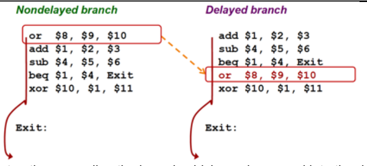
Branch Prediction. All branches are assumed to be not taken → we fetch the processor instruction and start pumping it through the pipeline stages. When the actual branch outcome is known, if we found out that it was guessed wrongly, we flush the successor instruction from the pipeline. Or else, if we had guessed correctly, there is no pipeline stall.

Branch is known to be **taken** in cycle 3. The 'and' instruction should not have been executed, and hence is flushed from the pipeline.



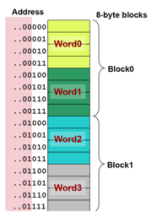
Delayed Branch. The branch outcome takes **X** number of cycles to be known, leading to **X** cycles stall. We move non-control dependent instructions (executed regardless of branch outcome) to the **X** slots following a branch (**branch-delay slot**).

In our MIPS processor with early branch, <u>branch-delay slot = 1</u>.

Best-Case Scenario	 Nondelayed branch: `or $8, $9, $10` `add $1, $2, $3` `sub $4, $5, $6` `beq $1, $4, Exit` `xor $10, $1, $11` `Exit:` Delayed branch: `add $1, $2, $3` `sub $4, $5, $6` `beq $1, $4, Exit` `or $8, $9, $10` `xor $10, $1, $11` `Exit:` There is an instruction preceding the branch which can be moved into the delayed slot.
Worst-Case Scenario	No such instruction can be found. Hence, we add a no-op (`nop`) instruction to the branch-delay slot.

L13 – Cache: Direct Mapped

Direct Mapped Cache



Cache Block / Line is a unit of transfer between memory and cache. Block size is typically more than 1 word.

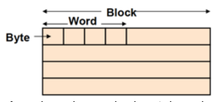
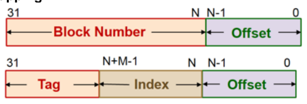


Figure: A cache, where a horizontal row is a block.

2^N-byte blocks are aligned at 2^N-byte boundaries in the main memory.

L

Mapping.



Memory Address Breakdown

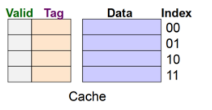
Cache Block size $= 2^N$ bytes
Number of Cache Blocks $= 2^M$

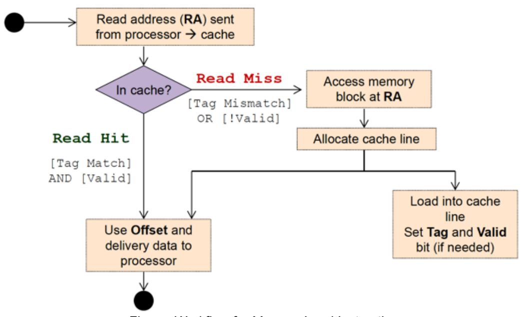
Offset $= N$ bits
Index $= M$ bits
Tag $= 32 - (N + M)$ bits

Item	Formula
Tag	$(Block\ Number) \,/\, (Number\ of\ Cache\ Blocks)$
Cache Index	$Cache\ Index = (Block\ Number)\ \%\ (Number\ of\ Cache\ Blocks)$ • Multiple Memory Blocks can be mapped to the same Cache Block → same **Cache Index**.

Table: Mathematical Formulae for Getting Tag and Index from Memory Address for DM Cache

Direct Mapped Cache Block Structure	

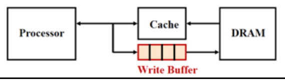

Valid Tag Data Index 00 01 10 11
Cache
	Every cache block contains a
	• **Tag:** A unique number.
	• **Valid Bit:** Indicating whether the cache line contains valid data.
	
	Cache Hit Condition:
	`(Valid[index] == TRUE) AND`
	`(Tag[index] == Tag[memory address])`



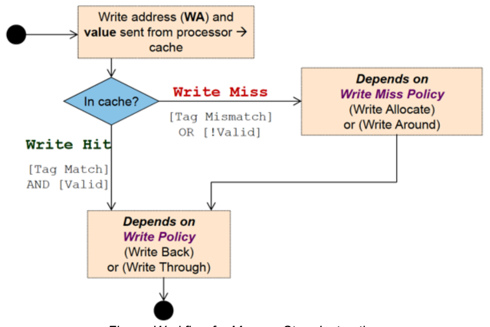
Read address (RA) sent from processor → cache
In cache?
Read Miss
[Tag Mismatch] OR [!Valid]
Access memory block at RA
Allocate cache line
Read Hit
[Tag Match] AND [Valid]
Load into cache line
Set Tag and Valid bit (if needed)
Use Offset and delivery data to processor

Figure: Workflow for Memory Load Instruction

Write Policy. Determines what you should do when writing data to a cache (since cache and main memory are inconsistent).

Write Policies	
Write-Through Cache	**Write-Back Cache**
Write data to both cache and memory. **Problem:** Write will operate at the speed of the main memory! **Solution:** Introduce a <u>write buffer</u> between the cache and main memory. The *processor* writes data to cache and write buffer, while the memory controller writes contents of the buffer to memory.  Processor Cache DRAM Write Buffer	Only write to cache. Write to main memory only when cache block is replaced (evicted). **Problem:** Quite 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 Policies	
Write Allocate	**Write Around**
Load the complete block into cache. Change only the required word in cache. Write to main memory depends on write policy.	Do not load the block to cache. Write directly to main memory only



Write address (WA) and value sent from processor → cache
In cache?
Write Miss
[Tag Mismatch] OR [!Valid]
Depends on Write Miss Policy (Write Allocate) or (Write Around)
Write Hit
[Tag Match] AND [Valid]
Depends on Write Policy (Write Back) or (Write Through)

Figure: Workflow for Memory Store Instruction

L14 – Cache: Direct Mapped

Parameter	Definition
Hit Rate	Fraction of memory accesses that hit (ie. Data is in cache)
Hit Time	Time to access cache. Hit time < Miss Penalty.
Miss Rate	$1 - Hit\ Rate$
Miss Penalty	Time to replace cache block and deliver data
Average Access Time	$Hit\ Rate \times Hit\ Time + (1 - Hit\ Rate) \times Miss\ Penalty$

Table: Various Parameters for Memory Access Time

Cache Miss Type	Definition	Solution
Compulsory / Cold Miss	First time a memory block is accessed.	Increase **cache block size**.
Conflict Miss	Two or more distinct memory blocks map to the same cache block.	Increase **cache size**. Use Set-Associative Caches.
Capacity Miss	Due to limited cache size.	

Table: Definitions for Cache Misses

Block Size Tradeoff.

Effects of Larger Block Size	
Advantages	Takes advantage of **spatial locality**.
Disadvantages	Larger **Miss Penalty** as it takes longer time to fill in the block. If block size is too big relative to cache size → too few cache block → **miss rate** will go up.

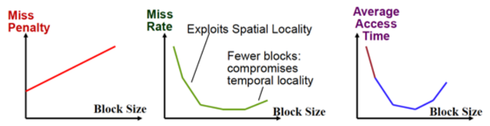
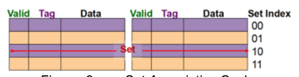
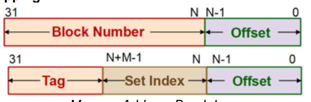


Figure: Effects of larger block size on memory access time

N-way Set Associative Cache.

Set Associative Cache Structure	
 Valid Tag Data Valid Tag Data Set Index 00 01 10 11 Set *Figure: 2-way Set Associative Cache*	The cache contains a number of sets, and each set contains **N** cache blocks. Each memory block maps to a unique cache set, and can be placed in any block within the set. We need to search for all blocks in the set to look for the memory block.

Mapping.



Memory Address Breakdown

Cache Block size $= 2^N$ bytes
Number of Cache Sets $= 2^M$

Offset $= N$ bits
Set Index $= M$ bits
Tag $= 32 - (N + M)$ bits

Item	Formula
Cache Index	$Cache\ Index = (Block\ Number)\ \%\ (Number\ of\ Cache\ Sets)$

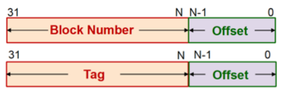
Table: Mathematical Formulae for Getting Tag and Index from Memory Address for SA Cache

Advantage of Associativity. A direct-mapped cache of size N has about the same miss rate as a 2-way set associative cache of size N / 2.

Fully Associative Cache.

In a 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 / cache set index. Instead, it can be placed in any location. On the flipside, we need to search all cache blocks for memory access.

Mapping.



Memory Address Breakdown

Cache Block size $= 2^N$ bytes
Number of Cache Blocks $= 2^M$

Offset $= N$ bits
Tag $= 32 - N$ bits
The block number serves as the tag.

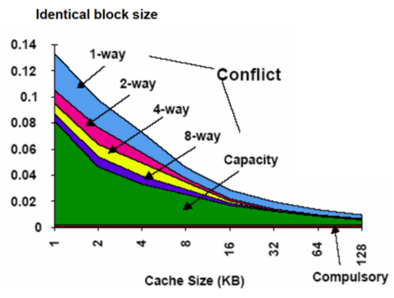
Cold Miss remains the same irrespective of cache size / associativity.

For the same cache size, **conflict miss** goes down with increasing associativity.

Conflict Miss is 0 for Fully Associative caches.

For the same cache size, **capacity miss** remains the same irrespective of associativity.

Capacity Miss decreases with increasing cache size.



$Total\ Miss = Cold\ Miss + Conflict\ Miss + Capacity\ Miss$
$Capacity\ Miss\ (FA) = Total\ Miss\ (FA) - Cold\ Miss\ (FA)$, when Conflict Miss → 0

Block Replacement Policy. For SA and FA Cache, we can choose where to place a memory block. This means potentially replacing another cache block if full.

Least Recently Used (LRU) Policy. 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. This exploits temporal locality. A drawback is that it is hard to keep track if there are many choices.

Other Replacement Policies.

- First-In-First-Out (FIFO)
- First-In-First-Out (FIFO), second chance variant
- Random Replacement (RR)
- Least Frequently Used (LFU)

L15 – Performance

Quantity	Formula
Performance	$Performance = \frac{1}{Response\ Time}$
Speedup n between x and y	$Speedup = \frac{Performance_x}{Performance_y} = \frac{ResponseTime_y}{ResponseTime_x}$
CPU Time Time spent executing lines of code in the program.	$CPU\ Time = \frac{Seconds}{Program} = \frac{Cycles}{Program} \times \frac{Seconds}{Cycle}$ $CPU\ Time = \frac{Instructions}{Program} \times \frac{Cycles}{Instruction} \times \frac{Seconds}{Cycle}$
Average Cycles-Per-Instruction (CPI)	$CPI = \frac{CPU\ Time \times Clock\ Rate}{Instruction\ Count} = \frac{Clock\ Cycles}{Instruction\ Count}$ $CPI = \sum_{k=1}^{n} CPI_k \times F_k$ where $F_k = \frac{I_k}{Instruction\ Count}$, I_k is the instruction frequency.

Table: Formulas related to the performance of a computer

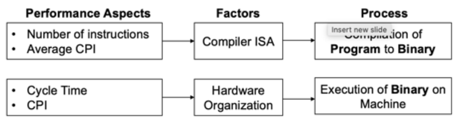
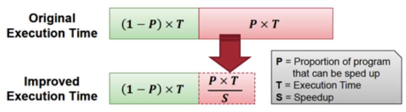


Figure: How Performance Aspects Affect Processes

Amdahl's Law. Performance is limited to the non-speedup portion of the program

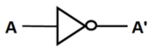
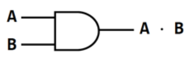
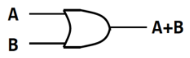
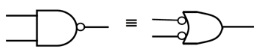
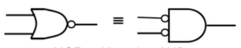
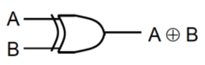
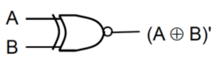
$$Speedup = \frac{1}{(1-P) + \frac{P}{S}}$$



L16 – Boolean Algebra

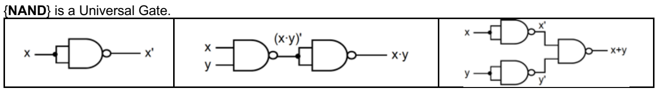
Boolean Algebra Laws and Theorems		
Identity Laws	$A + 0 = 0 + A = A$	$A \cdot 1 = 1 \cdot A = A$
Complement Laws	$A + A' = 1$	$A \cdot A' = 0$
Commutative Laws	$A \cdot B = B \cdot A$	$A + B = B + A$
Associative Laws	$A + (B + C) = (A + B) + C$	$A \cdot (B \cdot C) = (A \cdot B) \cdot C$
Distributive Laws	$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$	$A + (B \cdot C) = (A + B) \cdot (A + C)$
Idempotency	$X + X = X$	$X \cdot X = X$
Zero / One Elements	$X + 1 = 1$	$X \cdot 0 = 0$
Involution	$(X')' = X$	
Absorption	$X + X \cdot Y = X$	$X \cdot (X + Y) = X$
Absorption (variant)	$X + X' \cdot Y = X + Y$	$X \cdot (X' + Y) = X \cdot Y$
De Morgan's Theorem	$(X + Y)' = X' \cdot Y'$	$(X \cdot Y)' = X' + Y'$
	$(A + B + \dots + Z)' = A' \cdot B' \cdot \dots \cdot Z'$	
Consensus	$X \cdot Y + X' \cdot Z + Y \cdot Z = X \cdot Y + X' \cdot Z$	$(X + Y) \cdot (X' + Z) \cdot (Y + Z) = (X + Y) \cdot (X' + Z)$
Duality	If the **AND/OR** operators and identity elements **0/1** in a Boolean equation are interchanged, it remains valid. $X + 0 = X \Leftrightarrow X \cdot 1 = X$	

L17 – Logic Gates and Circuits

Gate	Diagram	Truth Table		
NOT		A		A'
		0		1
		1		0
AND		A	B	A · B
		0	0	0
		0	1	0
		1	0	0
		1	1	1
OR		A	B	A + B
		0	0	0
		0	1	1
		1	0	1
		1	1	1
NAND	 NAND ≡ Negative-OR	A	B	(A · B)'
		0	0	1
		0	1	1
		1	0	1
		1	1	0
NOR	 NOR ≡ Negative-AND	A	B	(A + B)'
		0	0	1
		0	1	0
		1	0	0
		1	1	0
XOR		A	B	A ⊕ B
		0	0	0
		0	1	1
		1	0	1
		1	1	0
XNOR		A	B	(A ⊕ B)'
		0	0	1
		0	1	0
		1	0	0
		1	1	1

Set of Universal Gates.
{**NOT**, **AND**, **OR**} gates are sufficient for building any Boolean function. {**NOR**} is a Universal Gate.

{**NAND**} is a Universal Gate.



For a Boolean function F,

To express it as a **sum of minterms**, find the summation of all minterms of the function where **output is 1**.

To express it as a **product of maxterms**, find the product of all maxterms of the function where **output is 0.**

x	y	z	F2
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

$$F2 = \sum m(1,4,5,6,7) = \prod M(0,2,3)$$

L18 – Simplification

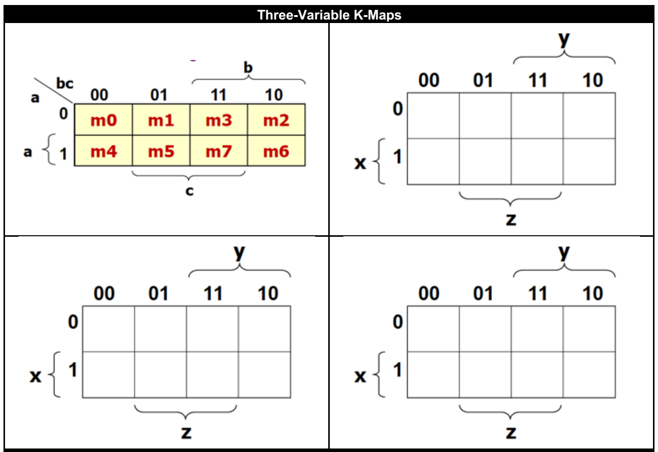
Term	Definition
Implicant	A product term that could be used to cover several minterms of the function.
Prime Implicant	The maximal (largest) possible implicant for a group of minterms.
Essential Prime Implicant	Prime Implicant that contains 1 or more unique minterm.

Table: Definition of terms in K-Maps

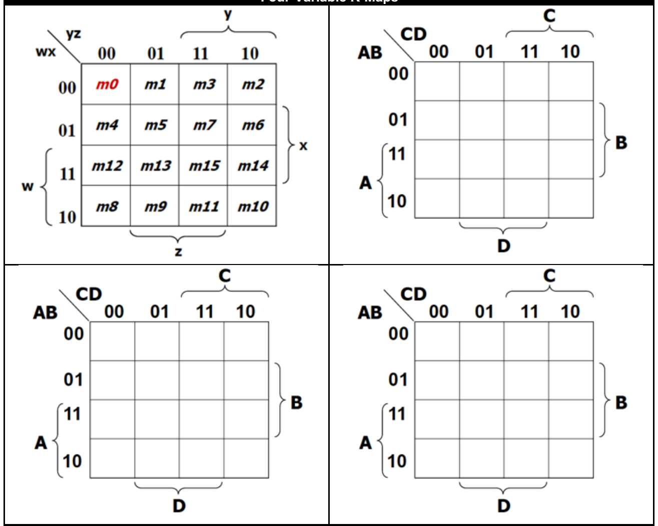
Simplification Algorithm. To find the simplified **Sum of Products** for a function, (1) Draw the Prime Implicant(s) for each minterm in the K-Map. (2) Using (1), take all Essential Prime Implicants. (3) Choose the smallest collection of Prime Implicant for the rest of minterms not covered in (2).

Finding Simplified POS. To find the simplified **Product of Sums** for a function F, (1) Find the simplified SOP expression for F', and then (2) Negate (1) to get the simplified POS expression of F.

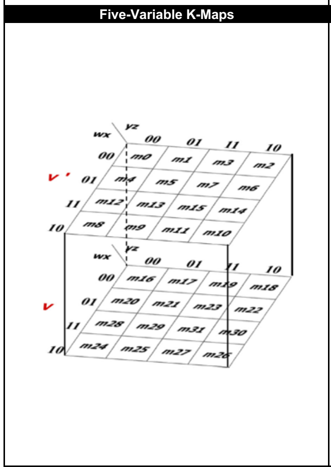
Three-Variable K-Maps



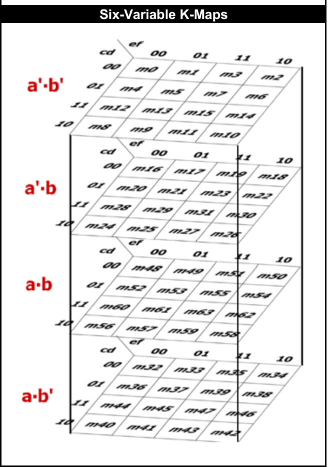
Four-Variable K-Maps



Five-Variable K-Maps



Six-Variable K-Maps

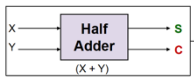


L19 – Combinational Circuits

Analysis of Combinational Circuits. (1) Label the inputs and outputs. (2) Obtain the functions of the immediate points and the outputs. (3) Draw the truth table. (4) Deduce the functionality of the circuit.

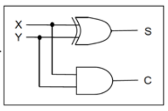
Gate-Level Design. (1) State Problem. (2) Determine and label the inputs and outputs of circuit. (3) Draw the truth table. (4) Obtain simplified Boolean functions. (5) Draw circuit diagram.

Half-Adder. Adds up only two bits.

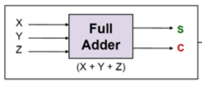


X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$C = X \cdot Y$$
$$S = X \oplus Y$$

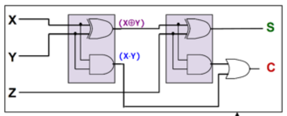


Full Adder. Note that **Z** refers to the carry in (to the current position), and **C** refers to the carry out (to the next position).



```
    1  1  1      Carry
 0  0  1  1      X
+0  1  1  1      Y
 1  0  1  0      S
```

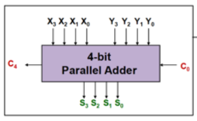
X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



$$C = X \cdot Y + (X \oplus Y) \cdot Z$$
$$S = X \oplus (Y \oplus Z) = (X \oplus Y) \oplus Z$$

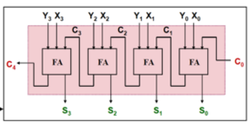
Block-Level Design. Decompose the main problem to sub-problems recursively, until sub-problem is small enough to be directly solved by blocks of circuits.

4-Bit Parallel Adder. Adds two 4-bit numbers together and a carry-in, to produce a 5-bit result. Carry is propagated by cascading the carry from one full adder to the next.

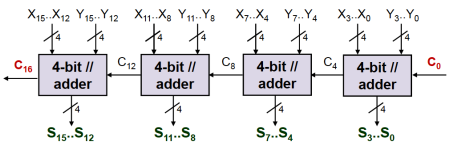


```
C4  C3  C2  C1  C0   Carry
    X3  X2  X1  X0   X
 +  Y3  Y2  Y1  Y0   Y
    S3  S2  S1  S0   S
```

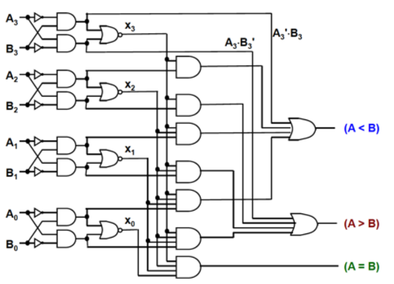
$$C_{i+1}S_i = X_i + Y_i + C_i$$
$$C_{i+1} = X_i \cdot Y_i + (X_i \oplus Y_i) \cdot C_i$$
$$S_i = X_i \oplus Y_i \oplus C_i$$



16-Bit Parallel Adder. Can be constructed from four 4-bit parallel adders.



Magnitude Comparator. Compares 2 values **A** and **B** to check if $A > B$, or $A = B$, or $A < B$.



To compare values $A = A_3A_2A_1A_0$ and $B = B_3B_2B_1B_0$, notice that:

If $((A_3 \cdot B_3')\ or\ (A_3 = B_3) \cdot (A_2 \cdot B_2')\ or\ \ldots)$, then A is larger.

Else if $((A_3' \cdot B_3)\ or\ (A_3 = B_3) \cdot (A_2' \cdot B_2)\ or\ \ldots)$, then B is larger.

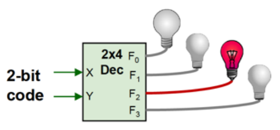
Else if $((A_3 = B_3)\ and\ (A_2 = B_2) \ldots)$ then A and B are the same.

Let $x_i = A_i \cdot B_i + A_i' \cdot B_i'$.

Circuit Delays. Given a logic gate with delay t. If inputs to the logic gate are stable at times $t_1, t_2, \cdots, t_n$, then the earliest time in which the output will be stable is $\max(t_1, t_2, \cdots, t_n) + t$. Repeat the rule for all gates to calculate the delays of all outputs of a combinatorial circuit.

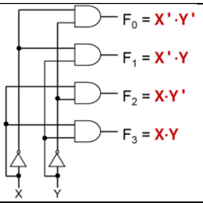
L20 – Medium Scale Integration (MSI) Components

Decoder



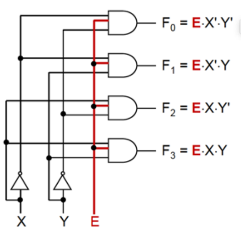
An $n \times m$ decoder takes the n bit code and converts it to one of the $m \le 2^n$ unique outputs. The output of a decoder corresponds to minterms.

1-enable: Use '1' to activate decoder.
0-enable: Use '0' to activate decoder.
Active-high: Output is '1' when selected.
Active-low: Output is '0' when selected.



X	Y	F_0	F_1	F_2	F_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

2×4 decoder without enable



E	X	Y	F_0	F_1	F_2	F_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	X	X	0	0	0	0

2×4 decoder with 1-enable

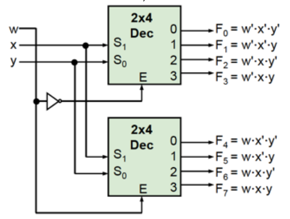
E	X	Y	F_0	F_1	F_2	F_3
0	0	0	1	0	0	0
0	0	1	0	1	0	0
0	1	0	0	0	1	0
0	1	1	0	0	0	1
1	X	X	0	0	0	0

2×4 decoder with 0-enable

Applications

Building Larger Decoder

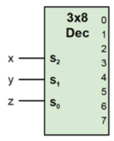
Larger decoders can be built from smaller decoders. Eg. A 3×8 decoder can be built from two 2×4 decoders with 1-enable, and an inverter.



Full Adder

A full adder can be implemented using a 3×8 decoder. Just implement the following truth table:

x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



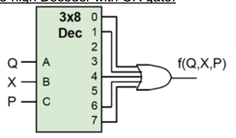
$$S(x,y,z) = \sum m(1,2,4,7)$$
$$C(x,y,z) = \sum m(3,5,6,7)$$

Implementing Functions

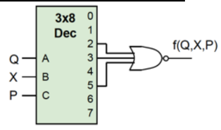
Functions can be implemented using a decoder and a suitable logic gate.

$$f(Q,X,P) = \sum m(0,1,4,6,7) = \prod M(2,3,5)$$

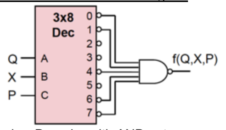
Active-high Decoder with OR gate:



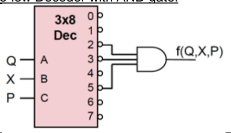
Active-high Decoder with NOR gate:



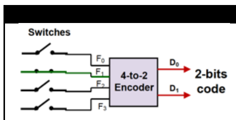
Active-low Decoder with NAND gate:



Active-low Decoder with AND gate:

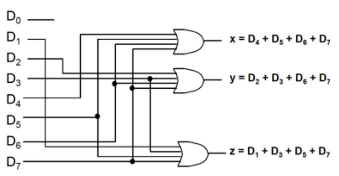


Encoder



An 2^n to n encoder codes the position of a single input line among 2^n possibilities as an n-bit code.

Example: An 8 to 3 decoder, with implementation and table shown below.



Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

There should be only one '1' among D_0 to D_7.

Priority Encoder

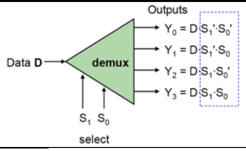
Suppose we assign priority to each input – If two or more inputs equal to 1, the input with the highest priority takes precedence. All inputs of 0 is considered to be invalid.

For example, the "compressed" truth table for a 4-to-2 priority encoder.

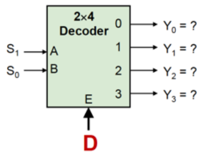
Note: For the '**X**'s in the **input**, they mean "for **both** 1 and 0", not don't care.

Inputs				Outputs		
D_0	D_1	D_2	D_3	x	y	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

Demultiplexer



Given one input data line, and N selection lines, a **demultiplexer** directs the input data to the selected output lines among 2^N possibilities.



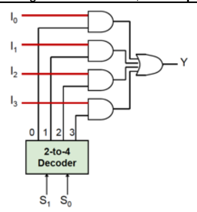
To implement a demultiplexer circuit, know that it is essentially identical to a decoder with enable.

S_1	S_0	Y_0	Y_1	Y_2	Y_3
0	0	D	0	0	0
0	1	0	D	0	0
1	0	0	0	D	0
1	1	0	0	0	D

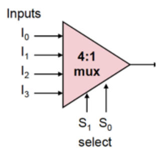
Truth table of a 1-to-4 Demultiplexer

Multiplexer

Using n selection lines, a **multiplexer** steers one of 2^n inputs to the output line.



A multiplexer can be implemented using a decoder.



S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

Truth table of a 4-to-1 multiplexer

Applications

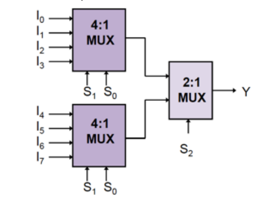
Building Larger Multiplexers

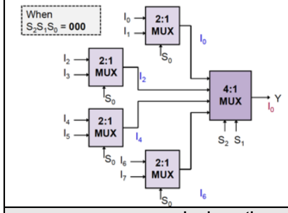
Implementing Functions

Using 2^n to 1 multiplexer.
To implement a function $F(A,B,C)$ using a multiplexer,
1. Express in sum-of-minterms form.
2. Connect n variables to the n selection lines.
3. Put a '1' on the data line if it is a minterm of the function, or '0' otherwise.

Joshua Chew Jian Xiang

An 8-to-1 multiplexer can be constructed with smaller multiplexers:

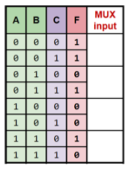




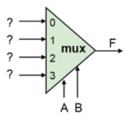
Using 2^{n-1} to 1 multiplexer.
1. Express in sum-of-minterms form.

$$F(A,B,C) = \sum m(1,3,5,6)$$

2. Reserve one variable (Eg. Least Significant Bit) for the input lines of the multiplexer, and use the rest for selection lines.
3. Use a truth table to group inputs by selection line values.

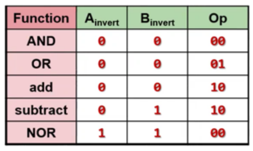


A	B	C	F	MUX input
0	0	0	1	
0	0	1	1	
0	1	0	0	
0	1	1	1	
1	0	0	0	
1	0	1	0	
1	1	0	1	
1	1	1	0	

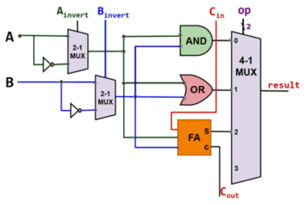


4. Determine multiplexer inputs by comparing input line and function output for corresponding selection line value groups.

Implementing an Arithmetic Logic Unit (ALU)



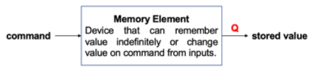
Function	A_{invert}	B_{invert}	Op
AND	0	0	00
OR	0	0	01
add	0	0	10
subtract	0	1	10
NOR	1	1	00



Instead of "choosing" the operation, we perform all operations and choose the desired output.

L21a – Sequential Logic: Building Blocks

Sequential Circuits. They are a type of logic circuit whose output depends on both present inputs and internal state (of the memory element). The same input can give a different output.

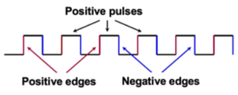

command
Memory Element
Device that can remember value indefinitely or change value on command from inputs.
Q
stored value

Command occurs at time t. $Q(t)$ refers to the current state, $Q(t+1)$ is the next state.

Command	$Q(t)$	$Q(t+1)$
Set	X	1
Reset	X	0
No change	0	0
	1	1

Figure: Characteristic table of a memory element

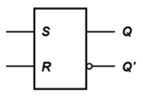
Clocking Signal. Usually a square wave.



Latches are pulse-triggered: ON = 1, OFF = 0

Flip-Flops are edge-triggered.
- *Positive edge-triggered:*
  ON = from 0 to 1; OFF = other time
- *Negative edge-triggered:*
  ON = from 1 to 0; OFF = other time

S-R Latch

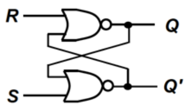


Inputs: **S** (SET), **R** (RESET)
Outputs: **Q** and **Q'**, complementary to each other.

- When **Q** = **HIGH** → Latch is in **SET** state.
- When **Q** = **LOW** → Latch is in **RESET** state.

Active-High S-R Latch (1 = active)

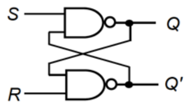
Constructed with cross-coupled NOR gates.



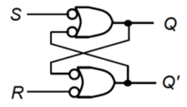
S	R	$Q(t+1)$	Description
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	X	Indeterminate

Active-Low S-R Latch (0 = active)

Constructed with cross-coupled NAND gates.

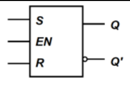


Alternatively, Negative-OR gates.

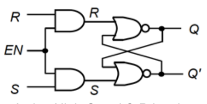


S	R	$Q(t+1)$	Description
1	1	$Q(t)$	No change
1	0	0	Reset
0	1	1	Set
0	0	X	Indeterminate

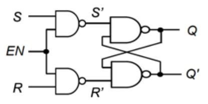
Gated S-R Latch



Output changes (if necessary) only when **EN** is high.

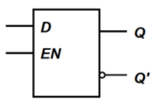


Active-High Gated S-R Latch



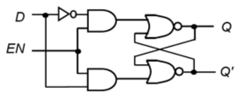
Active-Low Gated S-R Latch

Gated D Latch



A **gated D latch** makes input **R** equal to **S'**. This eliminates the undesirable condition of invalid state in the **S-R latch**.

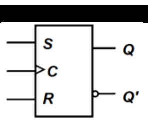
When **EN** is high, **Q** "follows" the **D** (data) input.
- When **D** = **HIGH** → Latch is in **SET** state.
- When **D** = **LOW** → Latch is in **RESET** state.



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

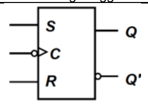
Shortcoming of Latches. Latches cannot satisfy the behaviour where: (1) we change value only at the start of a clock cycle, and then (2) maintain the value for the entire clock duration. We need this behaviour for Program Counters and Pipeline Registers.

S-R Flip-Flop



Positive edge-triggered

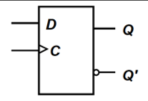
S	R	CLK	$Q(t+1)$	Description
0	0	X	$Q(t)$	No change
0	1	↑	0	Reset
1	0	↑	1	Set
1	1	↑	?	Invalid



Negative edge-triggered

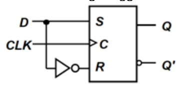
S	R	CLK	$Q(t+1)$	Description
0	0	X	$Q(t)$	No change
0	1	↓	0	Reset
1	0	↓	1	Set
1	1	↓	?	Invalid

D Flip-Flop



Positive edge-triggered

On the triggering edge of the clock pulse, **Q** "follows" **D** on the clock edge.
- When **D** = **HIGH** → **Q** becomes **HIGH** (**SET** state).
- When **D** = **LOW** → **Q** becomes **LOW** (**RESET** state).

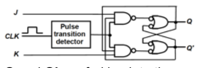


Converted from an S-R flip-flop by adding an inverter.

For a positive edge-triggered D Flip-Flop,

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

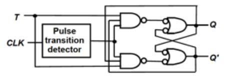
J-K Flip-Flop



Q and **Q'** are fed back to the pulse-steering NAND gates.

J	K	CLK	$Q(t+1)$	Description
0	0	↑	$Q(t)$	No change
0	1	↑	0	Reset
1	0	↑	1	Set
1	1	↑	$Q(t)'$	Toggle

T Flip-Flop



Formed by tying both inputs of the J-K Flip-Flop together.

T	CLK	$Q(t+1)$	Description
0	↑	$Q(t)$	No change
1	↑	$Q(t)'$	Toggle

L21b – Sequential Logic: Circuit Construction

Synchronous sequential circuits are those whose outputs change at specific time.

Building blocks. To build synchronous sequential circuits, we use Logic Gates (for combinational logic) and Flip-Flops (for memory element).

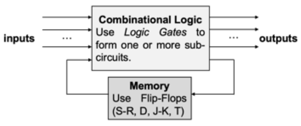
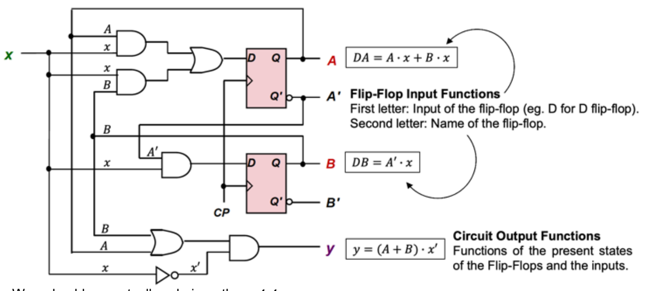

inputs
Combinational Logic
Use Logic Gates to form one or more sub-circuits.
outputs
Memory
Use Flip-Flops (S-R, D, J-K, T)

Figure: Overall structure of a sequential circuit

Analysis of Sequential Circuits. This is where you derive the behaviour from a given circuit diagram.

(1) Derive **equations** based on the circuit diagram provided.


x
A
x
x
B
B
A'
x
CP
B
A
x
x'
D Q
Q'
D Q
Q'
A
A'
B
B'
y
$DA = A \cdot x + B \cdot x$
Flip-Flop Input Functions
First letter: Input of the flip-flop (eg. D for D flip-flop).
Second letter: Name of the flip-flop.
$DB = A' \cdot x$
$y = (A+B) \cdot x'$
Circuit Output Functions
Functions of the present states of the Flip-Flops and the inputs.

We should eventually derive the **state equations** for the Flip-Flop inputs, and the **output functions** for the circuit outputs.

State Equations	Output Function
$A^+ = A \cdot x + B \cdot x$ $B^+ = A' \cdot x$	$y = (A+B) \cdot x'$

Take note that the **state equations** should be derived based on both the **characteristic table** of the Flip-Flops and the Flip-Flop Input Functions.

(2) Derive its **state table** based on the equations.

State Tables. Derived from the state equations and output functions, it consists of all possible binary combinations of <u>present states</u> and <u>inputs</u>. With m Flip-Flops and n inputs, the state table will have 2^{m+n} rows.

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

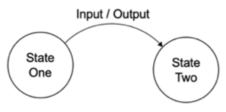
Full Table

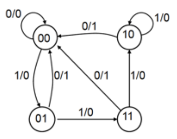
Present State	Next State		Output	
	$x=0$	$x=1$	$x=0$	$x=1$
AB	A^+B^+	A^+B^+	y	y
00	00	01	0	0
01	00	11	1	0
10	00	10	1	0
11	00	10	1	0

Compact Table

(3) Draw the **state diagram** based on the state table.

State Diagrams. An intuitive way to summarize the behaviour of a sequential circuit.





Design of Sequential Circuits. This is where you derive the logic circuit from a given set of specifications.

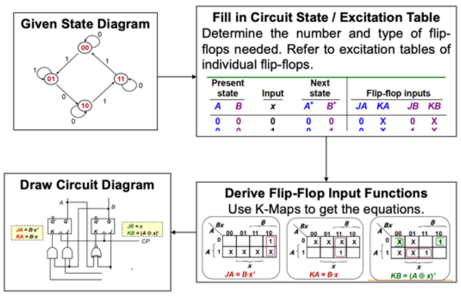

Given State Diagram
Fill in Circuit State / Excitation Table
Determine the number and type of flip-flops needed. Refer to excitation tables of individual flip-flops.
Draw Circuit Diagram
Derive Flip-Flop Input Functions
Use K-Maps to get the equations.
JA = B·x'
KA = B·x
KB = (A ⊕ x)'

Figure: Procedure for Designing Sequential Circuits

Excitation Tables. Given the required transition from present to next state, the table gives the required flip-flop input(s).

Flip-Flop Excitation Tables

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

J-K Flip-Flop

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

S-R 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