CS2100 Cheatsheet

C

Data Types – 32bit Processor
int – 32 bits
float – 32 bits
double – 64 bits
char – 8bit

Data Type Conversion
A op B -> C
If A and B are same data type, C is same data type
Else, lower data type and C is automatically promoted to match higher data type
e.g 2 / 5.0 -> 0.4, 2.0 / 5 -> 0.4

Assignment
int myInt;
myInt = 0.4 -> myInt stores 0
double myDouble;
myDouble = 2 -> myDouble == 2.0

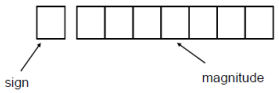
Type Casting
Promote/Demote a value by type casting
(double) 2 / 5 -> 0.4

Data Storage
Units:
Byte = 8 bits
Word = 4 bytes (32 bits) / 8 bytes (64bit), dependent on platform

N bits can represent up to 2^n values
32 bits -> 2^{32} values

Integer Representation
- Excess System
- Sign & Magnitude
Sign is represented by 'sign bit'
→ 0 for +
→ 1 for –

- Eg: a 1-bit sign and 7-bit magnitude format



- Largest value: 01111111 = $+127_{10}$
- Smallest value: 11111111 = -127_{10}
- Zeroes: 00000000 = $+0_{10}$, 10000000 = -0_{10}
- Range: -127_{10} to $+127_{10}$

- 1s-Complement System

Binary	n-bit 1s-Complement Representation
X (i.e. positive value)	X (no change)
-X (negative value)	$2^n - X - 1$

- For negative values, just flip the bits to get that value

Overflow

- Signed numbers are of a **fixed range**
- If the result of addition/subtraction goes beyond this range, an **overflow occurs**
- Overflow can be easily detected:
  - **positive** add **positive** → **negative**
  - **negative** add **negative** → **positive**

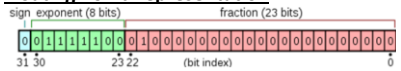
A+B/A-B (Addition/Subtraction)
Addition – Perform standard binary addition of A and B
Subtraction – Take 1s-complement of of B. Add 1s complement of B to A.

2-s Complement

Binary	n-bit 2s-Complement Representation
X (i.e. positive value)	X (no change)
-X (negative value)	$2^n - X$

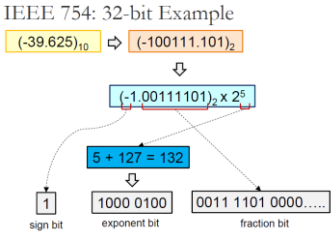
- To get negative values: from the back of the binary number, copy the digits until you meet the first 1. Flip all digits after the first 1

Floating Point Representation



IEEE-754
1 sign, 8 exponent, 23 mantissa
Sign: 0 = +ve, 1 = -ve
Exponent: Excess-127
Fraction: normalized to 1.X but take X only

IEEE 754: 32-bit Example



Characters
ASCII Code / Unicode
ASCII – 255 hard encoded mappings

C Pointers
int x;
int *ptr; (*pointer declaration)*
ptr = &x; (*address-of operator)*
x = *ptr; (*dereferencing)*

Parameter Passing
1. Pass By Value
2. Pass by address/Pass by pointer

Simple data types (int, float, char, double) and structures are **passed by value**
Arrays are **passed by address.**

Arrays
Int myArray[3] = { 1 };
- > [1, 0, 0]
- Array name by itself is the same as address of 0^{th} array element.

String
Character array to store multiple characters. Add special character ("\0") (ASCII value 0) to indicate it is a string.

Strutures

```
struct Fraction {
   int num;
   int den;
};
```

```
struct Fraction frac1 = { 1, 2 };

struct Fraction frac2 = { 5 };
```

- Structures are passed by value. A copy of structure is created when passed into function.

- To pass by pointer – use pointers.
*void printFrctn(struct Fraction *fptr)*
Structure: **Passed by address**

- As dereferencing a structure pointer and accessing a field is very common:
  - C provides a **shortcut** notation for this usage
  - The "->" is known as **indirect field selector**

```
void printFraction( struct Fraction *fptr )
{
    printf( "%d / %d", (*fptr).num,
                       (*fptr).den );
}
```

```
void printFraction( struct Fraction *fptr )
{
    printf( "%d / %d", fptr->num, fptr->den );
}
```

MIPS
32 registers

Name	Register number	Usage
$zero	0	Constant value 0
$v0-$v1	2-3	Values for results and expression evaluation
$a0-$a3	4-7	Arguments
$t0-$t7	8-15	Temporaries
$s0-$s7	16-23	Program variables

Name	Register number	Usage
$t8-$t9	24-25	More temporaries
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address

$at (register 1) is reserved for the assembler.
$k0-$k1 (registers 26-27) are reserved for the operation system.

Masking Operation – AND
→ Place **0s** into positions to ignore → bits turn into 0s
→ Place 1s for interested positions → bits remain same as original

Load lower 16 bits into register – *ORI*
→ Use lui + ori to load 32 bit value into register

MIPS Encoding
- R-format ***(op $r1, $r2, $r3)***
- I-Format ***(op $r1, $r2, immd)***
- J-Format ***(op Immd)***

R-Format

6	5	5	5	5	6
opcode	rs	rt	rd	shamt	funct

- Each field has a name:

rs- source register
rt- target register
rd- destination register
shamt – shift amt
opcode/funct – specifies instruction
→ add, sub, sll, srl etc

I-Format

6	5	5	16
opcode	rs	rt	immediate

- Again, each field has a name:

Immediate
- 16 bits, can represent constant up to 2^{16} values
- can be used to represent **signed** integer.
→ addi, andi, ori, xori, lui, lw, sw, bne, j

J-Format

6 bits	26 bits
opcode	target address

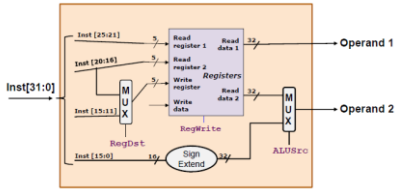
- As usual, each field has a name:

Optimization:
Jumps will only jump to word-aligned addresses, so last 2 bits are always *00.*
→ assume address ends with '00' and leave them out, we can now specify 2^{28} of 32-bit address!

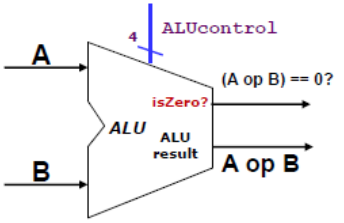
5-Stage MIPS Execution
Fetch → Decode & Operand Fetch → ALU → Memory Access → Result Write

Fetch Stage
- Use program counter (PC) to fetch instruction from memory

Decode Stage
Gather data from instruction fields, read opcode, read data from necessary registers.
→ In fetching read and write registers for register file, I-type instructions use multiplexer, r-type don't.



ALU-Stage
- Arithmetic Logic Unit
- Gets operands and operation from decode stage
- output result to memory stage
- Control:
- 4-bit to decide the particular operation
- IsZero – is the answer zero



Memory Stage
Input/Output depends on whether we are going to load or store from memory
Input:
- Memory Address
- Data to be written for store instructions
Control: Read vs Write
Output: Data read from memory if is load instruction

Result Write Stage
Route correct result into register file
Route result into 'Write data' of register file

MIPS Processor Controls

Control Signals

Control Signal	Execution Stage	Purpose
RegDst	Decode / Operand Fetch	Select the destination register number
RegWrite	Decode/Operand Fetch Result Write	Enable writing of register
ALUSrc	ALU	Select the 2nd operand for ALU
ALUControl	ALU	Select the operation to be performed
MemRead / MemWrite	Memory	Enable reading/writing of data memory
MemToReg	Result Write	Select the result to be written back to register file
PCSrc	Memory / Result Write	Select the next PC value

- Control Signals are generated based on instruction to be executed:
→ Opcode gives instruction format
→ For R-type, instruction has additional info: 6-bit funct code

RegDst – if coming from rd, this returns true (basically R-type format instructions).

ALUControl – Multilevel Decoding
- Use Opcode to generate 2-bit ALUop signal
→ Use ALUop signal and funct code to generate 4-bit ALUcontrol signal

SUB instruction (R-type):
Critical Path:
I-Mem →Reg.File → MUX(ALUSrc) → ALU → MUX(MemToReg) → Reg.File

LW instruction:
Critical Path:
I-Mem → Reg.File → ALU → DataMem → MUX(MemToReg) → Reg.File

BEQ instruction:
Critical Path:
I-Mem → Reg.File → MUX(ALUSrc) → ALU → AND → MUX(PCSrc)

Pipelining
Execution Stages – IF, ID, EX, MEM, WB
- Each ex. Stage takes 1 clock cycle, except updating of PC, WB of reg file

Datapath
- Idea: Data is used by same instruction in later stages
- Additional Regs in datapath called *pipeline registers*
-IF/ID, ID/IE, IE/EX, EX/MEM, MEM/WB

Corrected Datapath
- Reg in IF/ID latch not correct reg for WB
- Soln: Pass 'Write Reg' number throughout pipeline reg/latch

Pipelining: Control
Key Idea: Group control signals according to pipeline stage

Pipelining: Performance
- CT_{pipe} – max(Tk) + Td, where Tk is longest stage time among N stages, Td is pipeline overhead
- Cycles Needed – I + N – 1, N – 1 cycles to fill up pipeline
- Total Time needed for I instructions: $Time_{pipe}$ – Cycle x CT_{pipe} = (I + N – 1) x max(Tk) + Td

Ideal Speedup:
- Idea: Every stage takes same time
- No pipeline overhead
- I much bigger than N
→ Can also see this is how pipeline processor loses performance

Pipeline Hazards
- Structural, Data, Control hazards

Data Dependency: RAW
- When 1 instrcn needs mmr before data is correctly stored in that mmr

Solution: Data Forwarding
- Fwd data produced after EX stage by ALU to before ALU for next instruction
→ For load instrn, pipeline MUST store for 1 cycle as data is only produced in MEM stage (2 instrn before EX stage)

Control Dependency
Problem: Branching is only done originally in MEM (after isZero? calculated in ALU)
Solutions: Early Branch Res, Branch Predctn, Delayed Branching

Early Branching
- Make decision in Decode Stage
- Intro isZero? comparison component to be calculated in decode stage
=> Result known after decode stage → only need to store for 1 clock cycle!

Problem: If reg involved in comparison is produced by preceding instrn, i.e.
add **$s0**, $s1, $s2
beq **$s0**, $s3, exit
($s0 result only computed after EX, but $s0 result needed in beq in ID stage)

Solution: Add fwding path. However, 1 clock cycle delay still needed.
Total => 2 Clock cycle delays.
→ If **load** instrn followed by branch, result prdced after MEM stg, 2 clk cycle delay => 3 Clock cycle delays.

Branch Prediction
- Assume all branches not taken/taken
- If taken, flush pipeline
  - You only find out after EX stage
  → Waste 3 prior instructions

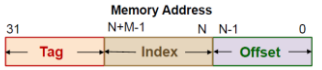
Delayed Branching
Idea: Move non-control dependent instrns into X slots following a branch
→ Delayed branching + early branching: Only 1 slot
-Must be non-control dependent instrn

Caching
- Keep frequently and used data in smaller but faster memory
Why does it work? **Principle of Locality**: Program accesses only small portion of memory address spaces within a small time interval

Types of Locality
- Temporal Locality
→ If an item is reference, it will tend to be referenced again soon
- Spatial Locality
→ If an item is referenced, nearby items will tend to be referenced soon

Processor check if data is in cache
→ If hit → Data returned to processor from cache
→ If miss → Data loaded from memory
→ Data placed in cache for future reference

Direct Mapped Cache



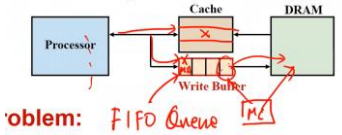
Cache Block divided into 3 components: Tag, Index, Offset

Offset – depending on **block size.**
→ 8 byte block => 3 bits offset (2^N byte block => n bits offset)
Index – 2^M number of cache blocks => M bit cache index
Tag – 32 – (N + M)

Cache Hit:
(Valid[index] == true) && (tag[index] == tag[memory address])
Basically tag will be unique for each different address that is mapped to the same block.

Memory Store Instctns (Write Policies)
Problem: When performing a store instruction, data in cache and memory becomes inconsistent.
→ Modified data only in cache and not in memory
Write-through Cache
→ Write data to cache and memory
vs **Write-back Cache**
→ Only write to cache, will write to memory when cache block evicted

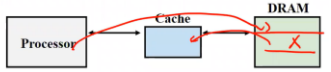
Write-through Cache



Problem: Operate at speed of memory if need to write to memory directly for every store instruction
Solution: Put write buffer between cache and main. Processor will write data to both cache + write buffer, **memory controller** will write from buffer to memory

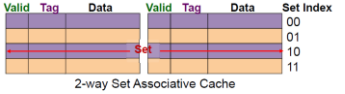
Write-back Cache
Problem: Wasteful if we write back every evicted cache block
Solution: Add dirty bit to each cache block. If cache contains overwritten data, change dirty bit to 1. When block is evicted, all blocks with dirty bit == 1 will be written into memory

Handling Cache Misses
Discussion: In a program, when a store word instruction is executed, the data arguably will not be needed in the near future.
Solution 1: Write allocate
→ Standard solution, load data from mmr to cache block, write back to mmr depending on write policy
Solution 2: Write around



→ Write directly to processor only

Set Associative (SA) Cache



Offset – same as DM
Set Index – Depends on number of **sets** instead of number of blocks
Tag – 32 – (N + M), refer to DM

Advantage of associativity
→ Reduce cold misses if mmr access sequence repeats: 0 4 0 4 0 4…
→ A DM cache of size N has about the same miss rate as a 2-way SA cache of size **N/2,** miss rate cuts a lot when go from DM to 2-way.

Fully Associative (FA) Cache
- Memory block placed in any location in cache
-Not restricted by cache/set index
-**But** need to search all blocks for mmr access

Cache Performance
→ Cold miss remains same regardless of cache associativity
→ Conflict miss reduces with increasing associativity
→0 Conflict miss for FA cache
→ For same cache size, capacity miss remains same regardless associativity
→ Capacity miss decreasing with increasing cache size

Total Miss = Cold + Conflict + Capacity

Block Replacement Policy
- LRU (Least Recently Used)
→ Record cache block that was accessed, choose block that was least recently accessed. (Temporal Locality)
Drawback: Hard to keep track if there are many choices
Other policies: FIFO, Random Replacement, Least frequently used

Performance
Performance = 1/response time
- Perf: bigger btr, resp time: smaller btr
- Speedup = $Perf_x$/ $Perf_y$
= $RespTime_x$ / $RespTime_y$
- Evaluate perf based on user CPU time

$$\text{CPU Time} = \frac{\text{Seconds}}{\text{Program}}$$

$$\text{CPU Time} = \frac{\text{Cycles}}{\text{Program}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

- Average Cycle Per Instrn (Avg CPI)

$$\text{CPI} = \sum_{k=1}^{n} \text{CPI}_k \times F_k \quad \text{where} \quad F_k = \frac{I_k}{\text{Instruction count}}$$

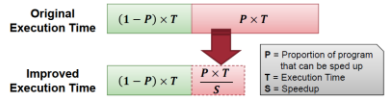
I_k = Instruction frequency

Idea: CPI of a particular instrn * how many times you use it

Influencing Factors of Performance
- Compilation, kind of instructions (MIPS etc)
→ Different compilers and its optimisation strats can generate different binary
- Hardware executed on machine (Cycle time, CPI)
→ Different cycle times (Diff clk freq) and CPIs (design of intrnl mechanism)

Boolean Algebra
→ Manipulation of algebraic variables, variables can only take values of 1s/0s

Amdahl's Law
- Performance is limited to the non-speedup portion of the program
- Corollary – Optimise the most common part 1st, not the slowest part!



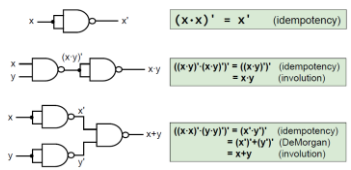
Basic Boolean Operations
→ NOT (~A, A')
→ AND (A.B, A ^ B)
→ OR (A+B, A v B)
NOT>AND>OR, or use parentheses to define priority

Duality → Completing flipping all AND/OR operators in a bool eqn allows eqn to still be valid (a+b).c <-> (a.b)+c
Two-for-one → If you prove 1 theorem, its dual form is also true

Boolean Functions
- Input k number of Boolean variables
- Output Boolean value
→ F1(x,y,z) = x'.y'.z + x.y'
→ Complement of function, F', obtain by flipping every operator/variable in expression

Logic Gates
- Fan-in – Number of inputs in a gate
Universal Gates
- Gates that can build any bool fn
- NOT/AND/OR gates sufficient for building any Boolean function
- NAND
- NOR

Implement **NOT** / **AND** / **OR** using only **NAND** gates



- Min-term (product term) vs Max-term(sum-term)
- Product of sum-term (POS) vs Sum of product-term (SOP)
- **Sum-of-minterms**
= the summation of all minterms of a function (**where output is 1**)
- **Product-of-maxterms**
= Product of all maxterms of the function (**where output is 0**)

$\Sigma m(1, 4, 5, 6, 7)$
$\Pi M(0, 2, 3)$
→ Notation for SOP (1,4,5,6,7) and POS(0,2,3)

Simplification
- Why simplify? Less logic gates used, cheaper, less power, faster
- Techniques: Algebraic, K-Maps, Quine-McCluskey

K-Maps
- Easy to use, limited to 5/6 variables
- Idea: Place **minterms** in a special arrangement so that simplifications can be easily performed

→ All values with difference of 1-bit to value in question are neighbors
I, PI, EPI
- Implicant: Product term used to cover several minterms of a fn
- Prime Implicant: Largest possible implicant for a group of minterms
- Essential Prime Implicant: PI that contains 1 or more unique minterm

Simplification Algorithm
1. Draw PIs for each minterm in K-map
2. Take all EPIs
3. Choose smallest collection of PI for minterms not covered in 2.

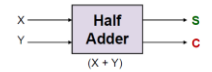
Don't Care Conditions
- In some problems, output not specified, can be 0 or 1
- Just like how we use Σm to represent minterms, we use Σd to represent don't-care minterms, i.e Σd(1,3,5)
→ Activate don't-care terms in K-map if it produces larger PIs

Combinational Circuits
- Output depends solely on present inputs for combinational circuits

Analysis
1. Label inputs and outputs
2. Obtain fns of intermediate pts and outputs
3. Draw truth table, deduce output

Design Methods
- Gate-Level vs Block-level Design
Steps for Gate-Level Design:
1. State Problem
2. Determine and label inputs and outputs of circuit
3. Draw Truth table
4. Obtain simplified Boolean functions
5. Draw logic diagram

Half Adder



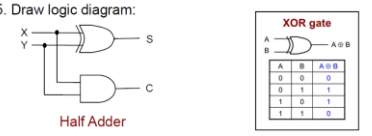
X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

3. Draw the truth table

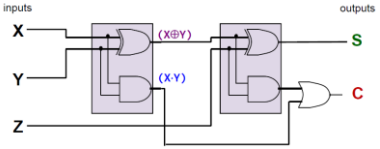
4. **Obtain simplified Boolean functions**
Example: C = X.Y
S = X'.Y + X.Y'
S can be further simplified to X⊕Y

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

5. Draw logic diagram:

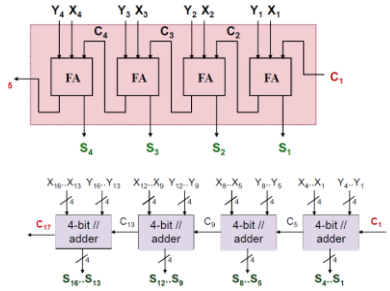


Full Adder
C = X·Y + (X⊕Y)·Z
S = X⊕(Y⊕Z) = (X⊕Y)⊕Z (XOR is associative)

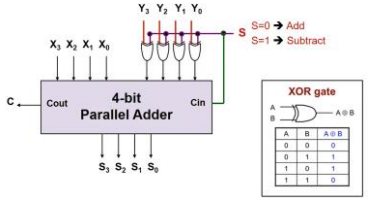


Steps for Block-Level Design:
1. Decompose main problem to sub-problems
2. Until sub-problems small enough to directly use blocks of circuits

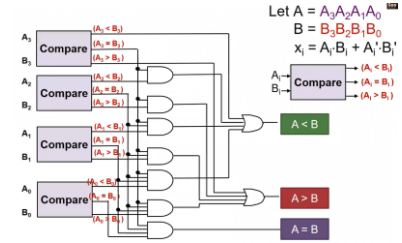
4-Bit, 16-Bit Parallel Adder



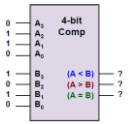
4-Bit Adder Cum Subtractor
→ Subtraction can be done via addition with 2s-complement numbers
→ Can be done by modifying the normal 4-bit parallel adder



Arithmetic Circuits
- A>B, A=B, A<B
Insight for 4-bit magnitude comparator

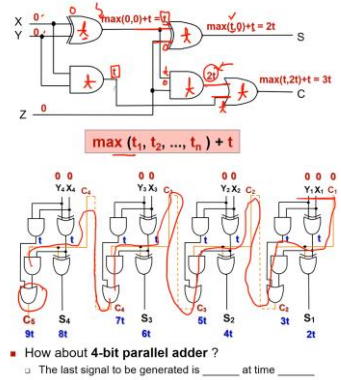


→ X_3, X_2, X_1, X_0 represents A3 == B3, A2 == B2 …
→ For A < B, A3'.B3 or (A3 == B3 and A2'.B2)…
→ For A > B, A3.B3' or (A3 == B3 and A2.B2')…
→ Basically compare bit by bit
- Block diagram of a 4-bit magnitude comparator



Circuit Delays

→ Assume all inputs available at time=0, assume each gate has delay t

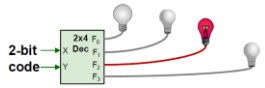


- Basically delay is a lot because output depends on previous outputs

Medium-Scale-Integration (MSI) Components

4 components: Decoder, Encoder, Multiplexer, Demultiplexer

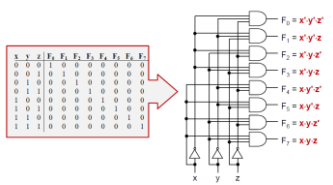
Decoder



- Depending on the **2-bit code**, one of the light bulbs will be **switched on**
- Truth table:

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

3 x 8 Decoder: Example



→ Key Idea: Output of decoder correspond to minterms.

- Lay out truth table, find out the F1 to Fn needed, connect based on sum of minterms (m1 + m2 + m5…)
- Easy to implement but may not be fast (internally many AND gates)

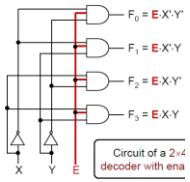
Decoder Variants

- Can be 1-enable or 0-enable
- Can be active- high/active-low (output is 1/0 when selected)

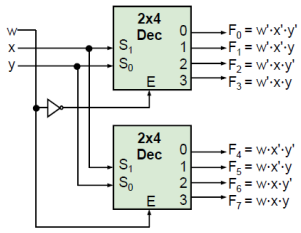
1-enable Decoder:

- Truth table:

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

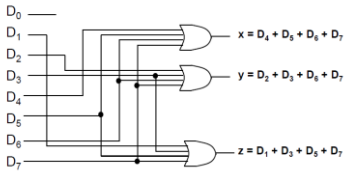


- Can build larger decoders from smaller decoders: two 2x4 to form one 3x8, two 3x8 to form 1 4x16



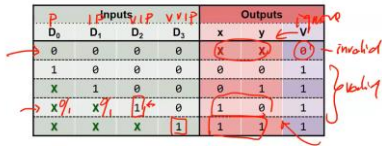
Encoder

- Converse of decoder



Priority Encoder

- **4-to-2 Priority Encoder**:

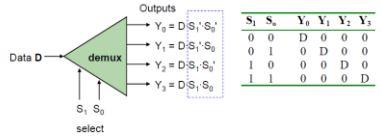


→ LSB takes largest precedence, so whichever first LSB is a 1, output it with a well-defined value.

→ Case 0000 still not covered, add a valid bit, if valid bit = 0 just ignore

Demultiplexer

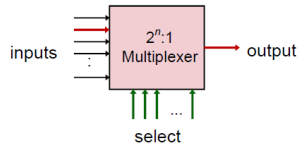
- Given 1 input data line, N selection lines, demux direct input data to the selected output line



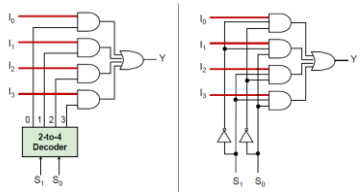
→ Exactly the same expression as decoder with enable

Multiplexer

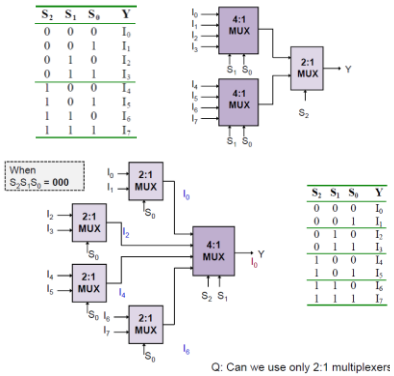
Given 2^n input lines, n selection lines, output input to output line



- Can construct with decoder



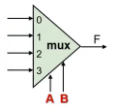
- Can construct larger multiplexers from smaller ones
- An **8-to-1 multiplexer** example:
  - Note the **placement of selector lines**



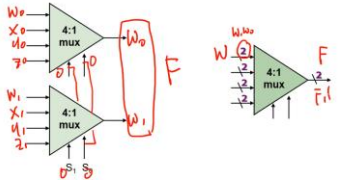
Implementing Functions: Mux

- Using smaller multiplexer (2 selector lines for 3 variable function)

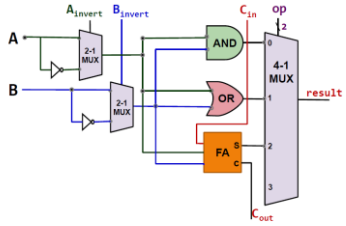
A	B	C	F	MUX Input
0	0	0	1	1
0	0	1	1	
0	1	0	0	C
0	1	1	1	
1	0	0	0	0
1	0	1	0	
1	1	0	1	C'
1	1	1	0	



Selecting multi-bit data



Implementing an ALU



Sequential Logic – Building Blocks

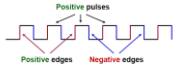
- Can be classified by timing

→ Synchronous: Outputs change only at specific time

→ Async: Outputs can change any time

- 2 types of triggers: Pulse-triggered/Edge-triggered
- Pulse triggered: Only react to changes during **positive pulses**
- Edge triggered: Only react in that tiny window that it can run its operation

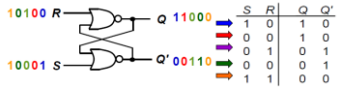
1. **Pulse-triggered**
   - Latches
   - ON = 1, OFF = 0



2. **Edge-triggered**
   - Flip-flops
   - 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

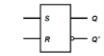
- Q = HIGH → Latch in set state, Q = LOW → Latch in reset state
- Active high input S-R Latch vs active low input S-R Latch (1 = active vs 0 = active)



S	R	Q	Q'
1	0	1	0
0	0	1	0
0	1	0	1
0	0	0	1
1	1	0	0

- Constructed with 2 NOR gates

S	R	Q	Q'	Action
0	0	NC	NC	No change. Latch remained in present state.
1	0	1	0	Latch SET.
0	1	0	1	Latch RESET.
1	1	0	0	Invalid condition.



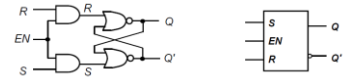
Active-low S-R Latch

Cross-coupled with NAND gates instead of NOR

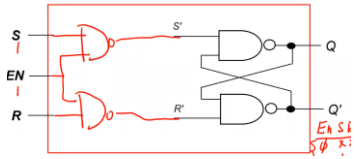
Opposite of Active-high SR Latch: R = 0, S = 1 → Reset, S = 0, R = 1 → Set state

Gated SR Latch

SR Latch + Enable input → Use 2 AND

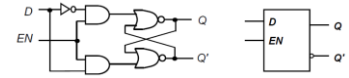


Active Low version – 2 NAND Gates



D-Latch

- Can come with enable
- Eliminates undesirable condition of invalid state in SR Latch

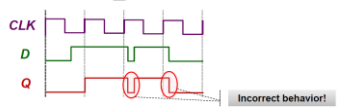


EN	D	Q(t+1)	
1	0	0	Reset
1	1	1	Set
0	X	Q(t)	No change

When *EN*=1, *Q(t+1) = ?*

Limitations

- We only want to change value at start of a clock cycle, maintain value for entire clock duration



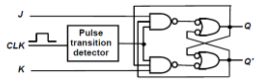
- D changes in middle of clock cycle, Q should change at start of next cycle

Flip-Flops

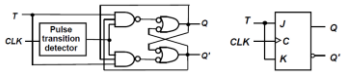
- Synchronous bi-stable devices

- Output changes state at rising (+ve) edge or falling (-ve) edge of clock cycle
→ SR Flip Flop, D FF, JK FF, T FF
- +ve and -ve edge-triggered flip-flops

- SR Flip-Flop
- D Flip-Flop
  - Can be converted from SR flip-flop by adding NOT gate
- J-K Flip-Flop
  - Q and Q' fed back to pulse-steering NAND gates



- T Flip-Flop
  - Tie J and K inputs together, essentially only has 0-0 or 1-1 inputs



Seq Circuits - Circuit Construction
Analysis Key Idea:
- Different from combinational circuits as same input can give different output
- Input & internal state dtrmine output
- Use of State Diagrams
- NEED to Derive State table
2. Derive state equation for FF inputs AND derive output functions for circuit outputs

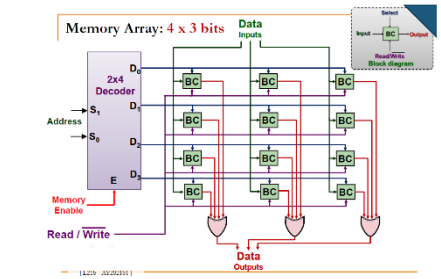
Sequential Circuit Design
- Aim: Derive logic circuit from given set of specifications, in the form of state table or state diagram
- Use excitation tables

1. Construct Excitation Tables
2. Use K-Map to derive functions for each input
3. Draw the circuit

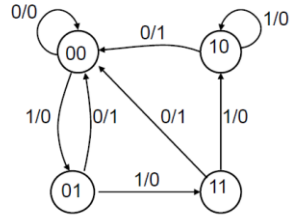
Computer Memory
- k address lines into input
- Control signal to indicate read/write – 1 to read, 0 to write
- For writing, supply n bit input
- For reading, n data output lines
- Enable signal as well

- Use D Flip Flop, use set to represent read and reset to represent write
- Add enable signal using AND gate
- Connect enable with AND to output as well to ensure no output regardless of state
- Directly input write data into R and S as well

- Data inputs – 1/0 to replace value in one bit cell(BC) or not
- Read/Write – 1/0 to read value or not
- output – go through or gate to output data. If read/write == 0 nothing to output
- Decoder – Input address, will output signal to corresponding array of BCs



Present State	Next State x=0	Next State x=1	Output x=0	Output 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



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

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

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

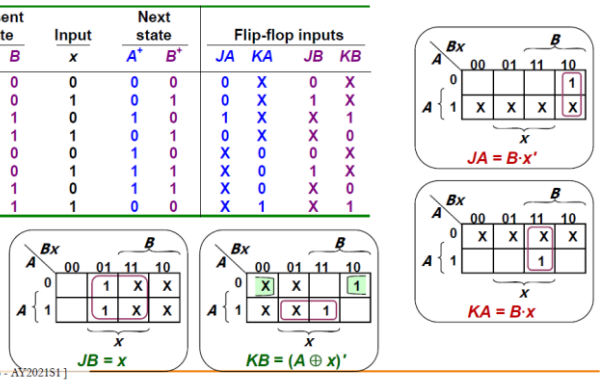
S	R	Q(t+1)	Comments
0	0	Q(t)	No change
0	1	0	Reset
1	0	1	Set
1	1	?	Unpredictable

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

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

- From state table, get flip-flop input functions.

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



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

JK Flip-flop

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

SR Flip-flop

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

D Flip-flop

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

T Flip-flop

Identity laws	
A + 0 = 0 + A = A	A · 1 = 1 · A = A
Inverse/complement laws	
A + A' = 1	A · A' = 0
Commutative laws	
A · B = B · A	A + B = B + A
Associative laws	
A + (B + C) = (A + B) + C	A · (B · C) = (A · B) · C
Distributive laws	
A · (B + C) = (A · B) + (A · C)	A + (B · C) = (A + B) · (A + C)

DeMorgan's	
(X + Y)' = X' · Y'	(X · Y)' = X' + Y'

*Can be generalized to more than two variables, e.g. (A + B + … + Z)' = A' · B' · … · Z'

Consensus	
X·Y + X'·Z + Y·Z = X·Y + X'·Z	(X+Y)·(X'+Z)·(Y+Z) = (X+Y)·(X'+Z)

Inverter (NOT gate)

A	A'
0	1
1	0

AND gate

A	B	A · B
0	0	0
0	1	0
1	0	0
1	1	1

OR gate

A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	1

a	b	c	F
0	0	0	m0
0	0	1	m1
0	1	0	m2
0	1	1	m3
1	0	0	m4
1	0	1	m5
1	1	0	m6
1	1	1	m7

a \ bc	00	01	11	10
0	a'.b'.c'	a'.b.c	a'.b.c	a'.b.c'
1	a.b'.c'	a.b'.c	a.b.c	a.b.c'

a \ bc	00	01	11	10
0	m0	m1	m3	m2
1	m4	m5	m7	m6

NAND gate

A	B	(A · B)'
0	0	1
0	1	1
1	0	1
1	1	0

NAND ≡ Negative-OR

NOR gate

A	B	(A + B)'
0	0	1
0	1	0
1	0	0
1	1	0

NOR ≡ Negative-AND

Idempotency	
X + X = X	X · X = X
Zero and One elements	
X + 1 = 1	X · 0 = 0
Involution	
(X')' = X	
Absorption	
X + X·Y = X	X·(X + Y) = X
Absorption (variant)	
X + X'·Y = X + Y	X·(X' + Y) = X·Y

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

Instruction Type	ALUop
lw / sw	00
beq	01
R-type	10

	EX Stage				MEM Stage			WB Stage	
	RegDst	ALUSrc	ALUop op1	ALUop op0	Mem Read	Mem Write	Branch	MemTo Reg	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

[L11 - AY2021S1]

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

XOR gate

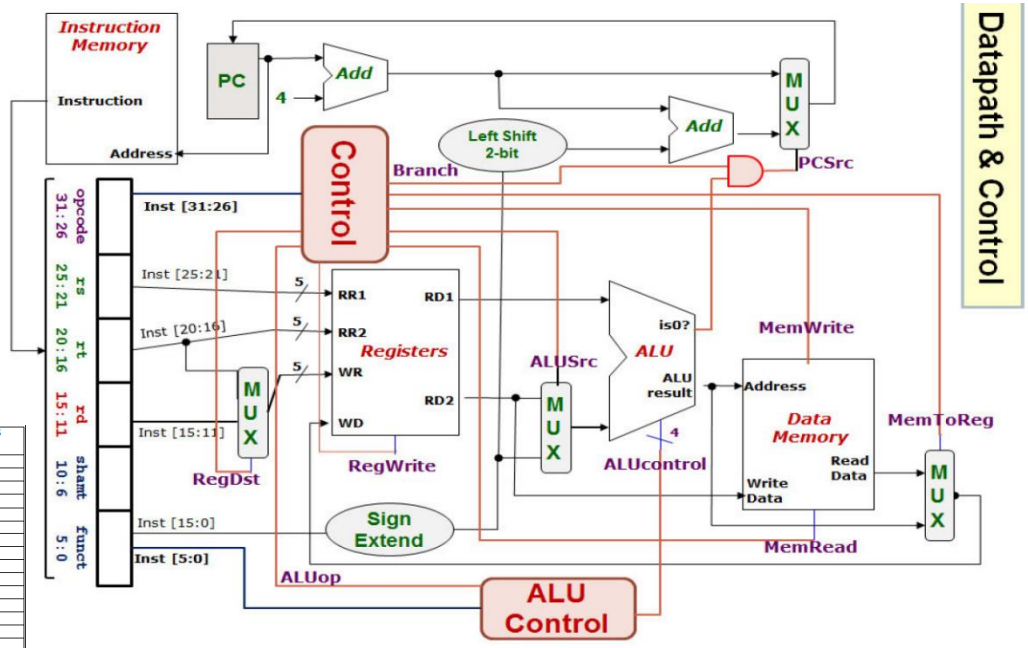
A	B	A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

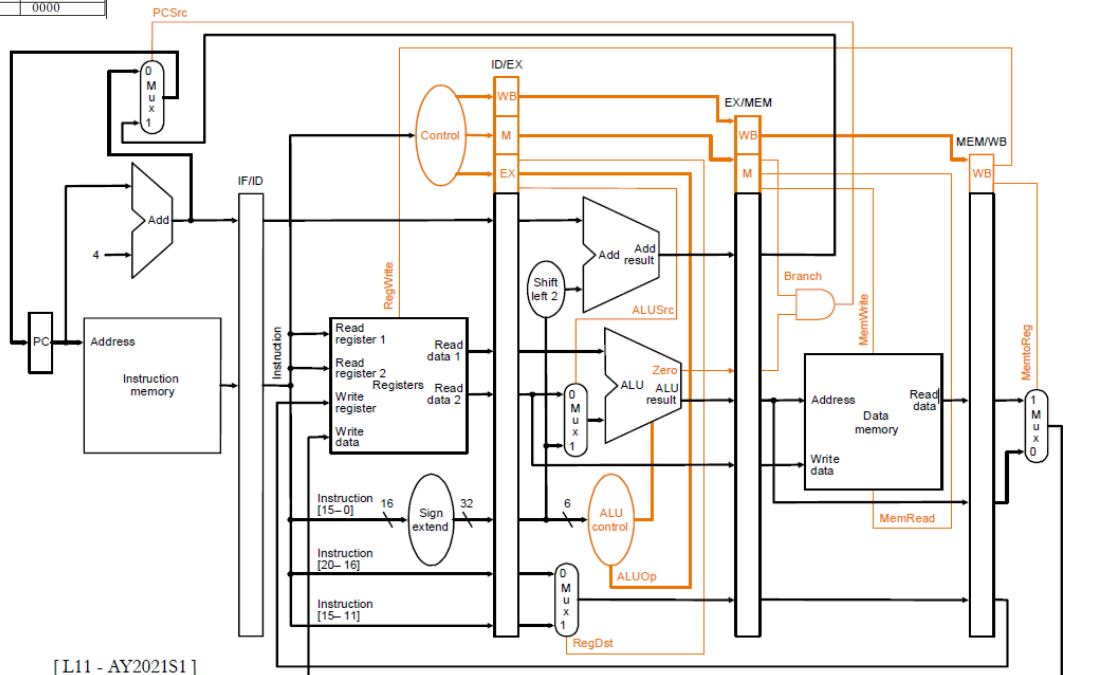
XNOR gate

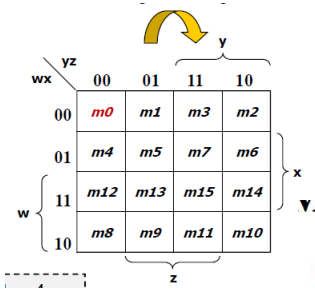
A	B	(A ⊕ B)'
0	0	1
0	1	0
1	0	0
1	1	1

XNOR can be represented by ⊙ (Example: A ⊙ B)

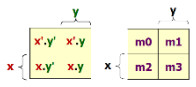
Value	Sign-&-Magnitude	1s Com-plement	2s Com-plement	Excess-8
+7	0111	0111	0111	1111
+6	0110	0110	0110	1110
+5	0101	0101	0101	1101
+4	0100	0100	0100	1100
+3	0011	0011	0011	1011
+2	0010	0010	0010	1010
+1	0001	0001	0001	1001
+0	0000	0000	0000	1000
-0	1000	1111	-	-
-1	1001	1110	1111	0111
-2	1010	1101	1110	0110
-3	1011	1100	1101	0101
-4	1100	1011	1100	0100
-5	1101	1010	1011	0011
-6	1110	1001	1010	0010
-7	1111	1000	1001	0001
-8	-	-	1000	0000

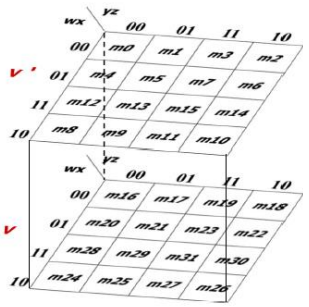


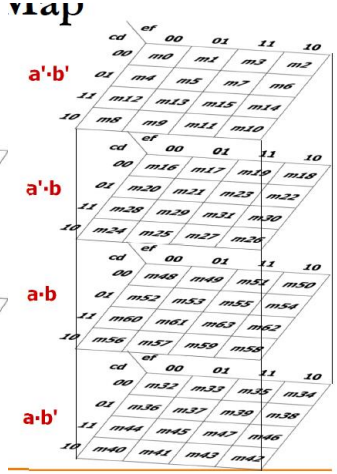




x	y	F
0	0	m0
0	1	m1
1	0	m2
1	1	m3







0x8df80000 = lw $24, 0($15); next PC = PC+4

Registers File				ALU		Data Memory	
RR1	RR2	WR	WD	Opr1	Opr2	Address	Write Data
$15	$24	$24	MEM([$15]+0)	[$15]	0	[$15]+0	[$24]

RegDst	RegWr	ALUSrc	MRd	MWr	MToR	Brch	ALUop	ALUctrl
0	1	1	1	0	1	0	00	0010

0x1023000C = beq $1, $3, 12; next PC = PC+4 or (PC+4)+(12×4)

Registers File				ALU		Data Memory	
RR1	RR2	WR	WD	Opr1	Opr2	Address	Write Data
$1	$3	$3 or $0	[$1]-[$3] or random value	[$1]	[$3]	[$1]-[$3]	[$3]

RegDst	RegWr	ALUSrc	MRd	MWr	MToR	Brch	ALUop	ALUctrl
X	0	0	0	0	X	1	01	0110

0x0285c822 = sub $25, $20, $5; next PC = PC+4

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

RegDst	RegWr	ALUSrc	MRd	MWr	MToR	Brch	ALUop	ALUctrl
1	1	0	0	0	0	0	10	0110