CS2100 Cheatsheet 17/18 Sem 2

by vig

Number Systems & Data Representation

Sizes of data/types

- byte : 8 bits
- nibble : 4 bits (half-byte)
- word : multiple bytes (1, 2, 4) (for MIPS it's 4)
- `int` : 4 bytes (1 bit for sign, 31 for magnitude)
- `float` : 4 bytes
- `double` : 8 bytes
- `char` : 1 byte

Representation & Complements

- Convert decimal whole numbers to base R : divide by R, first remainder is LSB, last is MSB
- Convert decimal fractions to base R : multiply by R, first carry is MSB, last is LSB
- base R to base R^N: partition in groups of N e.g groups of 4 for base 2 to base 16
- Convert to R-1s complement : Flip the digits; `digit = R - digit`
- Convert to Rs complement : Flip the digits, then add 1 to the number
- 1s complement has +ive and -ive 0
- 2s complement has only 1 representation of 0
- 2s complement can represent an ıadditional negative number e.g for binary, 1000 represents -8 (+8 cannot be represented in a signed 4 bit number)
- Convert to excess X: Take number minus X (0 refers to -x)
- IEEE 754 Floating-Point Representation: $sign|exponent|mantissa$
- Single-precision float has 1 bit sign, 8 bit excess-127 exponent, 23 bit mantissa (normalized with a leading bit 1 i.e the mantissa is the X in 1.X)
- Double has 1 bit sign, 11 bit excess-1023 exponent, 52 bit mantissa

Operations with binary numbers

- 2s complement addition: Simply add & ignore carry out of MSB
- 2s complement subtraction: take 2s complement of number to be subtracted, then do 2s addition.
- 1s complement addition: Add; If there is a carry out, add 1 to the result
- 1s complement subtraction: take 1s complement of number to be subtracted, then do 1s addition.
- check for **overflow: If result is opposite sign of both operands (that have the same sign)**

MIPS

R, I, J format

- **R**: $Opcode, rs, rt, rd, shamt, funct$
- **I**: $Opcode, rs, rt, Imm$
- rd is not used, check datasheet for instruction syntax
- For branch, Imm is the relative number of ıwords to go to (with respect to $PC + 4$), in 2s complement representation
- **J**: $Opcode, Address$
- First 4 bits are assumed to be 4 MSBs of PC+4. Last 2 bits assumed to be 0 (because of word addressing)

Instruction Set Architecture

Architectures & Endianness

- Von Neumann: Data(operands) stored in memory
- Stack: operands are on top of stack
- Accumulator: One operator is in the accumulator (a special register)
- Memory-memory (all operands in memory)
- Register-Register (all operands in registers) (MIPS)
- Big-endian: Most significant byte stored in lowest address
- Little-endian: Least significant byte stored in lowest address (easier to read)

Opcode encoding

- To maximize, reserve 1 instruction for lesser-bit instruction types.
- To minimize, reserve all but 1 instruction for lesser-bit instruction types
- Forumla for maximizing: $2^{no.of bits} * (1 - F)$ where F is the fraction of bits lost by reserving bits

Boolean Algebra Laws

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

Theorems

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

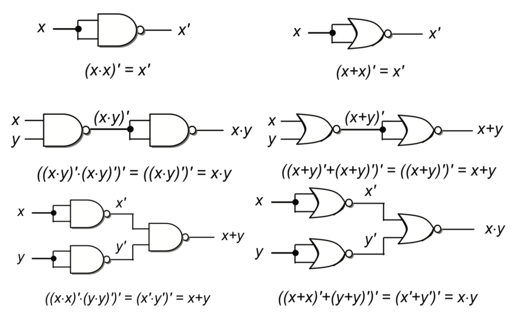
Minterms & Maxterms

- Sum-Of-Products (SOP): Product term or a logical sum of product terms
- $minterm$: Product term that contains n literals from all the variables
- Product-Of-Sum (POS): Sum term or a logical product of sum terms
- $Maxterm$: Sum term that contains n literals from all the variables
- $Mx = mx'$ because of De Morgan's
- Sum of 2 distinct Maxterms is 1 e.g $M1234 + M1120 = 1$
- Product of 2 distinct minterms is 0 e.g $m1234 \cdot m1120 = 0$

Combinatorial Circuits

Gates

- AND, OR, NOT is a complete set of logic
- NAND is a complete set of logic
- NOR is a complete set of logic
- Produce SOP with $AND >> OR$ or $NAND >> NAND$
- Produce POS with $OR >> AND$ or $NOR >> NOR$
- With negated outputs, use NAND to simulate OR and NOR to simulate AND

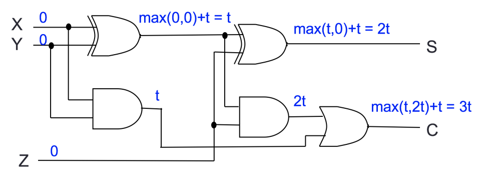


x
x'
(x·x)' = x'
x
x'
(x+x)' = x'
x
y
(x·y)'
x·y
((x·y)'·(x·y)')' = ((x·y)')' = x·y
x
y
(x+y)'
x+y
((x+y)'+(x+y)')' = ((x+y)')' = x+y
x
x'
y
y'
x+y
((x·x)'·(y·y)')' = (x'·y')' = x+y
x
x'
y
y'
x·y
((x+x)'+(y+y)')' = (x'+y')' = x·y

K-maps

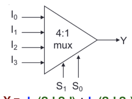
- Prime Implicant (PI) is a product term formed by combining the ımaximum possible no. of minterms (largest group)
- Essential Prime Implicant (EPI) is a PI that includes at least one minterm not covered by any other group
- Label the K-map rows/columns in a ıgray code manner e.g $00, 01, 11, 10$
- Grouping 2^N cells(only power-sizes are allowed) eliminates n variables
- EPIs are counted only by checking 1s, **not** Xs
- K-maps help to obtain canonical SOP, but might not provide the simplest expression possible (need to use boolean algebra for that)

Delays : Note that for combinatorial circuits, there is a delay: for every logic gate with n inputs, calculate $delay = max(t_1, t_2, \ldots t_n) + t_delay$



MSI Components

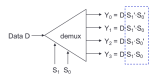
Multiplexer
Use minterm as selection line, using 0/1 as inputs. For smaller size multiplexer, use one of the variables for input lines.



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

$Y = I_0 \cdot (S_1' \cdot S_0') + I_1 \cdot (S_1' \cdot S_0) + I_2 \cdot (S_1 \cdot S_0') + I_3 \cdot (S_1 \cdot S_0)$

Demultiplexer



$Y_0 = D \cdot S_1' \cdot S_0'$
$Y_1 = D \cdot S_1' \cdot S_0$
$Y_2 = D \cdot S_1 \cdot S_0'$
$Y_3 = D \cdot S_1 \cdot S_0$

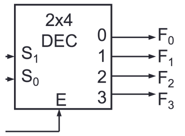
S_1	S_0	Y_0	Y_1	Y_2	Y_3
0	0	D	0	0	0
0	1	0	D	0	0
1	0	0	0	D	0
1	1	0	0	0	D

Encoder

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

$C_1 = F_2 + F_3$
$C_0 = F_1 + F_3$

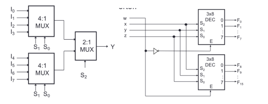
Decoder
Generate minterms and use OR to form a function Alternatively, use NOR on maxterms.



Priority Encoder

Inputs				Outputs		
D_0	D_1	D_2	D_3	f	g	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

Larger Components
Remove a decoder that gives duplicate outputs (w.r.t another decoder) by using an OR gate with the outputs from the first decoder, and the enable input of the second.



Sequential Logic

Excitation Tables

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

(a) *S-R* flip-flop.

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

(b) *J-K* flip-flop.

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

(c) *D* flip-flop.

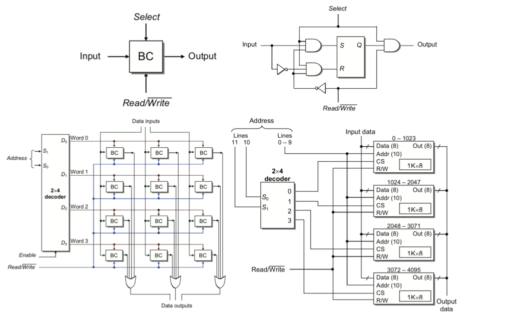
Q	Q^+	T
0	0	0
0	1	1
1	0	1
1	1	0

(d) *T* flip-flop.

- For m flip-flops, up to 2^m states exist.
- SR has invalid code while JK uses that for the toggle code

- T is the ıtoggle flip-flop
- D is the ısetting flip-flop
- Negative input for *Clock* → flip-flop is negative edge-triggered

Static RAM



- Dyanmic RAM does not use flip-flop as cells
- For BC, Write is 0, Read is 1
- 1K*8 RAM ⇒ 1024words*8bits
- In 12 bit address to 4K*8 RAM constructed using 1K*8 blocks, the 2 most significant bits are fed into decoder to determine which block to use.
- Expand horizontally to increase word size, vertically to increase memory size

Pipelining

Pipeline register contents

- IF/ID: Instruction from memory & $PC + 4$
- ID/EX: Data read from regsiter files, 32-bit Sign extended Imm, & $PC + 4$
- EX/MEM: Imm, & $(PC + 4) + (Imm * 4)$, ALU result, $isZero$ signal & $RD2$ from register file
- MEM/WB: ALU result, Memory read data & write regsiter data (passed through all pipelines)

Performance

- If cycle/clock time is given, just use that
- Single cycle:
  $CT_{seq} = \sum_{k=1}^{N} T_k$
  $Time_{seq} = I * CT_{seq}$ (choose the maximum CT_{seq})
- Multi-cycle [1 stage per cycle, cycle time chosen to be time for longest stage]
  $CT_{multi} = max(T_k)$ i.e longest stage time
  $Time_{multi} = I * AverageCPI * CT_{multi}$
- Pipeline [Several stages per cycle]
  $CT_{pipeline} = max(T_k) + T_d$ where T_d is the pipeline register overhead
  $Time_{pipeline} = (I + N - 1) * CT_{pipeline}$
- If $N_{intstructions} >> N_{stages}$,
  $Speedup_{pipeline} = \frac{Time_{seq}}{Time_{pipeline}}$

Hazard and resolution

- Without data forwarding: If dependent cycle is
  · right before: 2 cycle delay
  · 2 cycles before: 1 cycle delay
- With data forwarding: If dependent cycle is
  · dependent on lw: 1 cycle delay
  · otherwise: no delay
- Without control measures: 3 cycle delay
- With early branching/resolution: 1 cycle delay after branch instruction
  · with forwarding & dependent on non-lw: 1 cycle bef branch
  · with forwarding & dependent on lw: 2 cycles bef branch
  · without forwarding: dependent: 2 cycle delay bef branch
- With branch prediction:
  · 3 cycles occur if no early branching
  · 1 cycle occur if there is early branching
  · then, instructions either get flushed/not flushed
- With delayed branch: If ∃ instruction before branch that can be moved into delayed slot, move it. Else, stall/no-op

Cache

Average Access time

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

Direct Mapped Cache

- Blocks in cache: 2^M
- Bytes per block: 2^N

Set Associative Cache

- N-way SAC → N cache blocks per set
- Bytes per block: 2^M
- Cache bocks = $\frac{Size_{cache}}{Size_{block}}$
- Sets = $\frac{CacheBlocks}{N} = 2^N$

Fully Associative Cache

- Bytes per block: 2^N

For each address

- Set Index = $(val \bmod 2^{N+M})//2^N$
- Word Index = $(val \bmod 2^N)//Bytes_{word}$
- Tag = $val//2^{N+M}$

Miss Rates

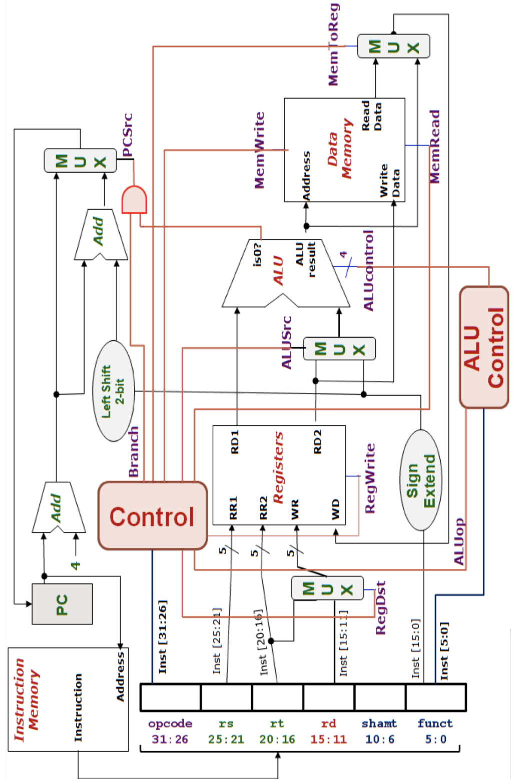
- Conlfict miss rates decrease with increasing associativity
- DMC of size N has the same miss rate as a 2-way SAC of size $\frac{N}{2}$
- Capacity miss only depends on cache size, same size → same capcatiy miss
- As cache size increases, capcacity miss decreases

Block Replacement

- Least Recently Used: Note that it is hard to keep track if there are many choices and there is a cost to keeping track of this as well
- First in First out
- Random Replacement
- Least Frequently Used

Writing Policy

- Write through cache: Write to both cache and main memory
- Write back cache: Only write to cache, write to memory when block is replaced
- Write Miss – Write allocate: Load complete block and write onto the cache ⇒ Write to main memory if using write through policy
- Write Miss – Write around: Do not load block to cache, write to memory only



4-bit system

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

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

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

	opcode	operand	operand
Type-A	6 bits	5 bits	5 bits

	opcode	operand
Type-B	11 bits	5 bits

Max (1 type A) = $1 + (2^6 - 1) * 2^5$
Min (1 type B) = $(2^6 - 1) + 2^5$

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