CS2100 Computer Organisation

AY 24/25 Sem 2 — github/omgeta

1. Number Systems

Weighted-positional number systems with base-R, has each position weighted in powers of R.

i. N positions give R^N values
ii. M values need $\lceil \log_R M \rceil$ positions

Conversion

Base-R→Decimal: repeated multiply by R^{n-1}

Decimal→Binary (Generalised for Base-R):

i. Whole numbers: repeated division by 2, $LSB \rightarrow MSB$
ii. Fractional: repeated multiplication by 2, $MSB \rightarrow LSB$

Binary→Octal: partition in groups of 3

Octal→Binary: convert each digit to 3 bits

Binary→Hexadecimal: partition in groups of 4

Hexadecimal→Binary: convert each digit to 4 bits

C Programming

ASCII Chars are represented with 7 bits and 1 parity bit

`char` is 1 byte, `int` + `float` are 4 bytes, `long` + `double` are 8 bytes.

Functions only modify `struct` if passed as pointer or in array with: `(*object_p).a = 1;`, `object_p->a = 1;`

Arrays in `struct`s are deep-copied (but not pointers to arrays).

Negative Numbers

Sign-and-Magnitude:

i. MSB is the sign bit (0 is positive)
ii. Range: $-(2^{N-1}-1)$ to $2^{N-1}-1$
iii. Negation: Flip MSB

1s Complement (useful for arithmetic):

i. Negated x, $-x = 2^N - x - 1$
ii. Range: $-(2^{N-1}-1)$ to $2^{N-1}-1$
iii. Negation: Invert bits
iv. Overflow: add carry to result, wrap around
v. $(b-1)s$: $-x = b^N - x - 1$

2s Complement (useful for arithmetic):

i. Negated x, $-x = 2^N - x$
ii. Range: -2^{N-1} to $2^{N-1}-1$
iii. Negation: Invert bits, then add 1
iv. Overflow: truncate
v. bs: $-x = b^N - x$

Excess-M (useful for comparisons):

i. Start at $-M = -2^{N-1}$, for N bits.
ii. Range: -2^{N-1} to $2^{N-1}-1$
iii. Express x in Excess-N: $x + N$

Real Numbers

Fixed-Point (limited range):

i. Reserve bits for whole numbers and for fraction, converting with 1s or 2s complement

IEEE 754 Floating-Point (more complex):

i. Sign 1-bit (0 is positive)
ii. Exponent 8-bits (excess-127)
iii. Mantissa 23-bits (normalised to $1.m \times 2^{exp}$)

single-precision: 1-bit sign / 8-bit exponent / 23-bit mantissa

2. ISA

Instruction Set Architecture (ISA) defines instructions for how software controls the hardware.

von Neumann Architecture:

i. Processor: Perform computations.
ii. Memory: Stores code and data (stored memory).
iii. Bus: Bridge between processor and memory.

Storage Architectures:

i. Stack: Operands are implicitly on top of the stack.
ii. Accumulator: One operand is implicitly in the accumulator.
iii. Memory-Memory: All operands in memory.
iv. Register-Register: All operands in registers.

Endianness (ordering of bytes in a word):

i. Big Endian: MSB in lowest address
ii. Little Endian: LSB in lowest address

Instruction Length:

i. Fixed-Length: easy fetch and decode, simplified pipelining, instruction bits are scarce
ii. Variable-Length: more flexible

Instruction Encoding, under fixed-length instructions involves extending opcode for instruction types with unused bits:

1. Minimise: maximise opcode range of instruction type with least opcode bits
2. Maximise: minimise opcode range of instruction type with least opcode bits

Shortcut for Maximise given $A > B > C$ instruction types, where $|X|$ is the minimum number of instruction $X = 2^A - |B|2^{A-B} - |C|2^{A-C} + |B| + |C|$

3. MIPS

MIPS Assembly Language:

i. Mainly Register-Register, Fixed-Length
ii. 32 registers, each 32-bits (4-byte) long
iii. 2^{30} words contains 32-bits (4-byte) long
iv. Memory addresses are 32-bits long

Arithmetic Instructions:

i. $C16, C5$ are 16, 5 bit patterns
ii. $C16_{2S}$ is a sign-extended 2's complement
iii. NOT: `nor` with `$zero`, or `xor` with all 1s
iv. Large constants: `lui` (31:16 bits) + `ori` (15:0 bits)

Operation	Opcode in MIPS	Meaning
Addition	`add $rd, $rs, $rt`	`$rd = $rs + $rt`
	`addi $rt, $rs, C16`$_{2s}$	`$rt = $rs + C16`$_{2s}$
Subtraction	`sub $rd, $rs, $rt`	`$rd = $rs - $rt`
Shift left logical	`sll $rd, $rt, C5`	`$rd = $rt << C5`
Shift right logical	`srl $rd, $rt, C5`	`$rd = $rt >> C5`
AND bitwise	`and $rd, $rs, $rt`	`$rd = $rs & $rt`
	`andi $rt, $rs, C16`	`$rt = $rs & C16`
OR bitwise	`or $rd, $rs, $rt`	`$rd = $rs | $rt`
	`ori $rt, $rs, C16`	`$rt = $rs | C16`
NOR bitwise	`nor $rd, $rs, $rt`	`$rd = $rs ↓ $rt`
XOR bitwise	`xor $rd, $rs, $rt`	`$rd = $rs ^ $rt`
	`xori $rt, $rs, C16`	`$rt = $rs ^ C16`

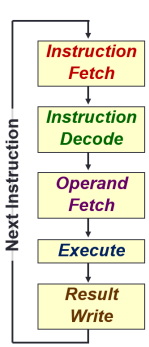
Memory Instructions:

i. `lw $rt, offset($rs)`: load contents to `$rt`
ii. `sw $rt, offset($rs)`: store contents to `$rs`
iii. Use `lb, sb` for loading bytes (like chars)

Control Instructions:

i. `beq/bne $rs, $rt, label/imm`: jump if $=/\neq$.
ii. Branch: adds `imm x 4` to `$PC = $PC + 4` on branch. Range: $\pm 2^{15}$ instructions
iii. `j label/imm`: jump unconditionally.
iv. Pseudo-direct: takes 4 MSBs from `$PC + 4`, and 2 LSB omitted since instructions are word-aligned. Range: 256-byte boundary of `$PC + 4` 4 MSBs
v. `slt $rd, $rs, $rt`: set `$rd` to 1 if `$rs < $rt`

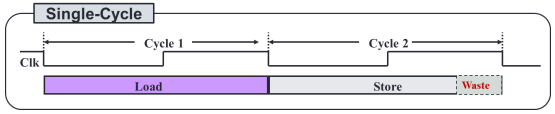
Datapath



1. **Fetch:**
   - Get instruction from memory
   - Address is in **P**rogram **C**ounter (PC) Register
2. **Decode:**
   - Find out the operation required
3. **Operand Fetch:**
   - Get operand(s) needed for operation
4. **Execute:**
   - Perform the required operation
5. **Result Write (Store):**
   - Store the result of the operation

Clock:

i. Single-Cycle: Read + Compute are done within clock period. PC Write on rising clock edge. Time taken depends on longest instruction.



ii. Multicycle: Break up execution into multiple steps (e.g. fetch, decode, execute, memory, writeback), with each step taking one cycle. Time taken depends on slowest step.

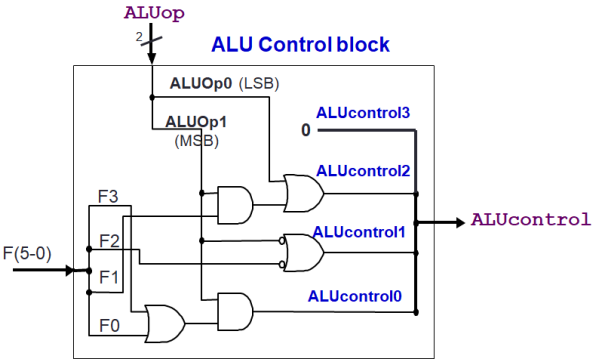
Critical Paths:

i. R-type/Immediate:
Inst Mem→RegFile→MUX (ALUSrc)
→ALU→MUX (MemToReg)→RegFile
ii. `lw/sw`:
Inst Mem→RegFile→MUX
(ALUSrc)→ALU→DataMem→MUX
(MemToReg)→RegFile
iii. `beq`:
Inst Mem→RegFile→MUX
(ALUSrc)→ALU→AND→MUX (PCSrc)

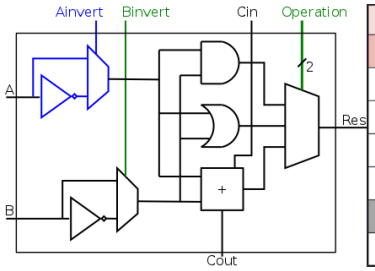
Control

Control Signals:

i. `RegDst` @Decode: Selects destination register.
(0) → `$rt` (1) → `$rd`
ii. `RegWrite` @Decode: Enable register write.
(0) → no write (1) → `WD` written to `WR`
iii. `ALUSrc` @ALU: Second 2nd ALU operand.
(0) → `RD2` (1) → `SignExt(Imm)`
iv. `ALUcontrol` @ALU: Second ALU operation.
2-bit `ALUop` from opcode + funct for 4-bits
v. `MemRead` @Memory: Enable memory read.
(0) → no read (1) → read `Addr.` into `Read Data`
vi. `MemWrite` @Memory: Enable memory write.
(0) → no write (1) → write `Write Data` into `Addr.`
vii. `MemToReg` @Writeback: Select writeback result.
(0) → `ALU result` (1) → Memory `Read data`
viii. `Branch` @Memory: Select next `$PC`.
(0) → `$PC + 4` (1) → `($PC + 4) + (imm x 4)`



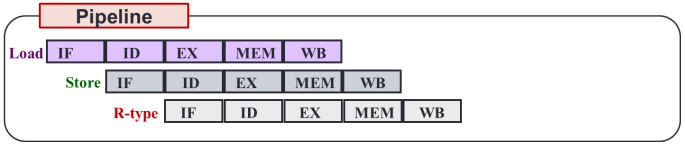
1-bit ALU



ALUcontrol			Function
Ainvert	Binvert	Operation	
0	0	00	AND
0	0	01	OR
0	0	10	add
0	1	10	subtract
0	1	11	slt
1	1	00	NOR

Pipelining

Pipelining improves throughput of workload by running instructions in parallel, with each pipeline stage taking one cycle with no overlapping of same stage.



Pipelining Stages:

i. **IF** (Fetch): Fetch instruction, `PC + 4`
ii. **ID** (Decode): Opcode, register file read
iii. **EX** (Execute): ALU execution or branch calc
iv. **MEM** (Memory Access): `lw/sw` access memory
v. **WB** (Writeback): Write result to register file

Pipeline Registers:

i. **IF/ID:** Instruction, PC+4
ii. **ID/EX:** RD1/2, Imm, PC+4, WriteReg
iii. **EX/MEM:** ALU result, (PC+4)+(imm×4), RD2, WriteReg
iv. **MEM/WB:** ALU result, Mem Read Data, WriteReg

Performance

Time for I instructions, for T_k stage time, under N stages:

i. Single-Cycle:
- $CT_{seq} = max(\sum_k^N = 1T_k)$
- $Time_{seq} = I \times CT_{seq}$

ii. Multi-Cycle:
- $CT_{multi} = max(T_k)$
- $Time_{multi} = I \times AverageCPI \times CT_{multi}$

iii. Pipeline:
- $CT_{pipeline} = max(T_k) + t_d$, with overhead t_d
- $Time_{pipeline} = Cycles \times CT_{pipeline}$
- Ideal Cycles $= I + N - 1$

Ideally, speedup over single-cycle is $\frac{Time_{seq}}{Time_{pipeline}} \approx N$

Hazards

Structural: Conflict in shared resource (e.g. MEM)

Data: Data dependencies (e.g. read-after-write; RAW)

i. Forwarding: EX/MEM or MEM/WB → EX
ii. Stalling: For `lw` delay

Control: Branch taken not known till EX stage, worst case 3-cycle stall

i. Early branch: Branch check at ID for 1-cycle stall
ii. Prediction: Assume all branches not taken, flush pipeline if wrong
iii. Delayed branch: Move instructions executed regardless of branch outcome into branch delay slots (1 per stall)

Forwarding Paths

RAW:

i. EX/MEM→EX
ii. EX/MEM→MEM (`xx`→`sw`)

Memory:

i. MEM→MEM (`lw`→`sw`)

Early Branch:

i. EX/MEM→ID

Common Delays

Without forwarding:

Instruction	Delay
RAW	+2
Load Word	+2
Branch at MEM	+3
Branch at ID	+1

With forwarding:

Instruction	Delay
RAW	+0
Load Word	+1
Branch at MEM	+3
Branch at ID	+1
RAW + Branch at ID	+1
Load + Branch at ID	+2

4. Boolean Algebra

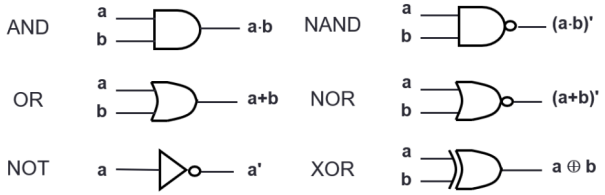
Standard forms:

i. Literal: boolean variable or it's complement
ii. Product term: logical product (AND) of literals
iii. Sum term: logical sum (OR) of literals
iv. Sum-of-Products: logical sum of product terms
v. Product-of-Sums: logical product of sum terms

Canonical forms:

i. Minterm (of n variables): prod. term of n literals. Denoted $m0$ to $m[2^n - 1]$; e.g. $m3 = x' \cdot y \cdot z$
ii. Maxterm (of n variables): sum term of n literals; Denoted $M0$ to $M[2^n - 1]$; e.g. $M3 = X + Y' + Z'$
iii. Sum-of-Minterms: canonical SOP expression e.g. $\sum m(0, 1, 2, 3) = m0 + m1 + m2 + m3$
iv. Product-of-Maxterms: canonical POS expression e.g. $\Pi M(4, 5, 6, 7) = M4 \cdot M5 \cdot M6 \cdot M7$

Logic Gates



Complete sets of logic are sufficient for any boolean function: {AND, OR, NOT}, {NAND}, {NOR}

SOP Circuit: 2-level AND-OR, or 2-level NAND.
POS Circuit: 2-level OR-AND, or 2-level NOR.

Karnaugh Maps

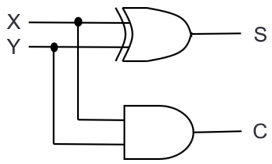
Implicants are product terms covering minterms:

1. Prime implicant (PI): implicant of maximum size
2. Essential prime implicant (EPI): PI including minterm not covered by any other PI

5. Combinational Circuits

Combinational circuit outputs depend only on inputs.

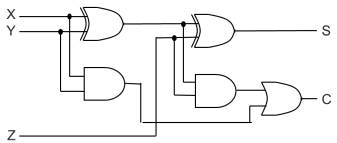
Half Adder



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

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

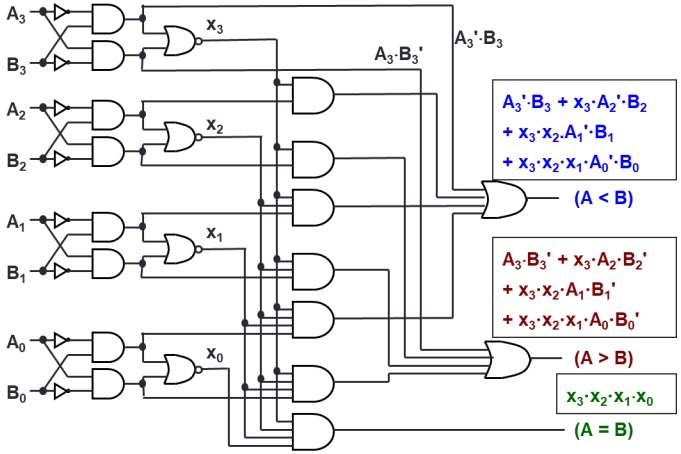
Full Adder



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

Magnitude Comparator

Let $A = A_3A_2A_1A_0, B = B_3B_2B_1B_0; x_i = A_i \cdot B_i + A_i' \cdot B_i'$



Circuit Delays

Inputs with delays $t_1, \cdots, t_n$ into gate with delay t will have delay $max(t_1, \cdots, t_n) + t$.

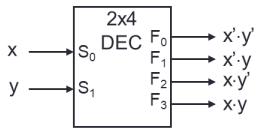
Delay of n-bit parallel adder, t is delay of full adders:

$$S_n = ((n-1)2+2)t, \quad C_{n+1} = ((n-1)2+3)t$$

MSI Components

Decoders (n to 2^n)

Output line are minterms, and circuits can be represented with sum of decoder outputs.

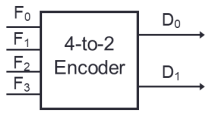


E	X	Y	F_0	F_1	F_2	F_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	d	d	0	0	0	0

Decoder with 1-enable

Encoders (2^n to n)

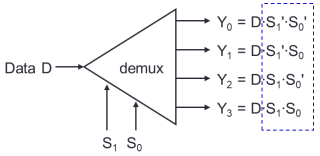
Priority encoders allow multiple inputs on, with highest priority taking precedence.



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

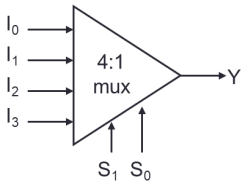
4-to-2 Priority Encoder

Demultiplexer (data decoders)



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

Multiplexer (data encoders)



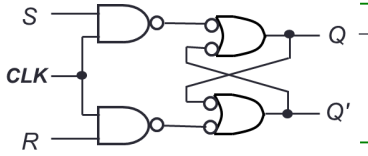
I_0	I_1	I_2	I_3	S_1	S_0	Y
d_0	d_1	d_2	d_3	0	0	d_0
d_0	d_1	d_2	d_3	0	1	d_1
d_0	d_1	d_2	d_3	1	0	d_2
d_0	d_1	d_2	d_3	1	1	d_3

6. Sequential Circuits

Sequential circuit outputs depend on inputs and state:

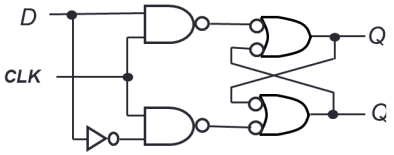
i. Latches pulse-triggered; flip-flops edge-triggered
ii. Gated circuits are enable-input
iii. Self-correcting if unused states → used states

SR Flip-Flop



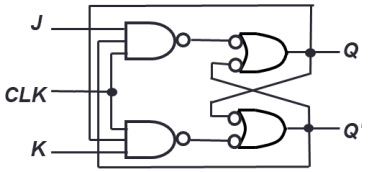
S	R	CLK	Q(t+1)
0	0	X	Q(t)
0	1	↑	0
1	0	↑	1
1	1	↑	?

D Flip-Flop



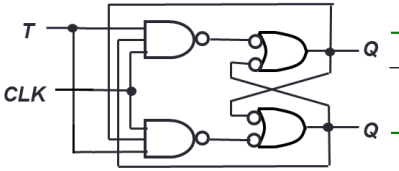
D	CLK	Q(t+1)
1	↑	1
0	↑	0

JK Flip-Flop



J	K	CLK	Q(t+1)
0	0	↑	Q(t)
0	1	↑	0
1	0	↑	1
1	1	↑	Q(t)'

T Flip-Flop



T	CLK	Q(t+1)
0	↑	Q(t)
1	↑	Q(t)'

Excitation

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

7. Cache

Caches reduce frequency of slow direct memory access in DRAM (slow & cheap) by using SRAM (fast & expensive).

i. Hit: data in cache
   - Rate: fraction of accesses that are hits
   - Time: time to access cache data

ii. Miss: data not in cache
   - Penalty: time to replace cache block + hit time

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

Principle of Locality:

i. Temporal: Same data likely re-accessed soon.

ii. Spatial: Nearby data likely be assessed soon.

Types of Misses:

i. Cold: Due to never being previously allocated.

ii. Conflict: In direct mapped/set associative cache; due to previous eviction by different block.

iii. Capacity: In fully associative cache; due to previous eviction as a result of small cache size.

Policies

Read Miss Policy:

i. Load data into cache, then from cache to register

Write Policy:

i. Write-through: Write to both cache and memory.

ii. Write-back: Write to cache and set dirty bit; write to memory on block eviction and reset dirty bit

Write Miss Policy (data to write not in cache):

i. Write Allocate: Cache data and follow write policy.

ii. Write Around: Write directly to memory only.

Block Replacement Policy:

i. Least Recently Used (LRU; most common)

ii. First In First Out (FIFO)

iii. Random Replacement (RR)

iv. Least Frequently Used (LFU)

Addresses can be grouped by block number:

i. 2^N bytes in block $\rightarrow N$ offset bits

Direct Mapped Cache

Blocks mapped to location index in cache:

i. 2^M cache blocks $\rightarrow M$ index bits

ii. Block stored with identification tag and valid bit

k-way Associative Cache

Blocks mapped to k locations at set index in cache:

i. 2^M cache sets (blocks / k) $\rightarrow M$ set index bits

ii. Block stored with identification tag and valid bit

Fully Associative Cache

Blocks placed anywhere but need to search all blocks

Performance

Direct mapped cache of size N has almost same miss rate as 2-way associative cache of size $N/2$.

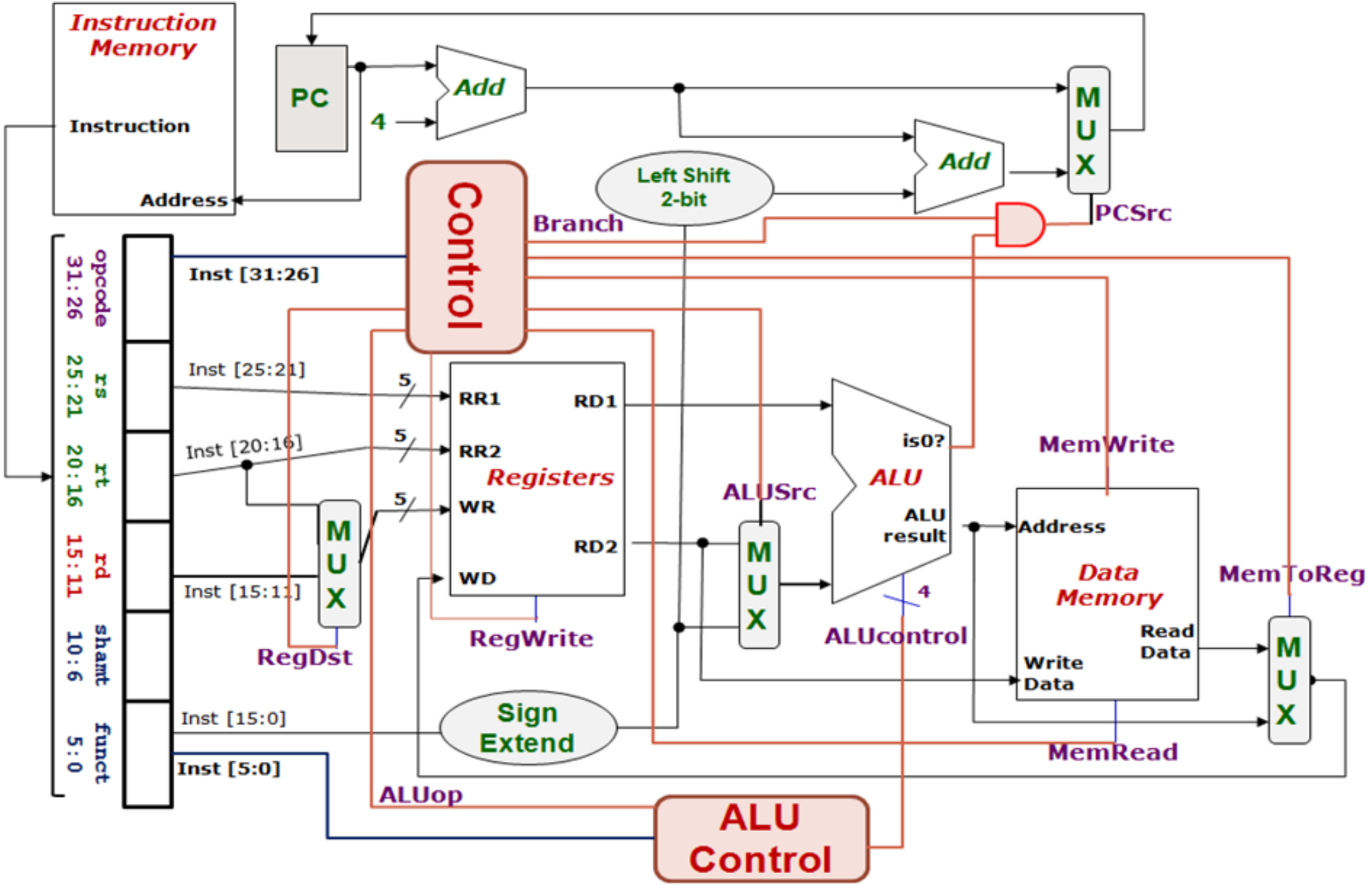
Miss Efficiencies:

i. Cold miss does not depend on size/associativity

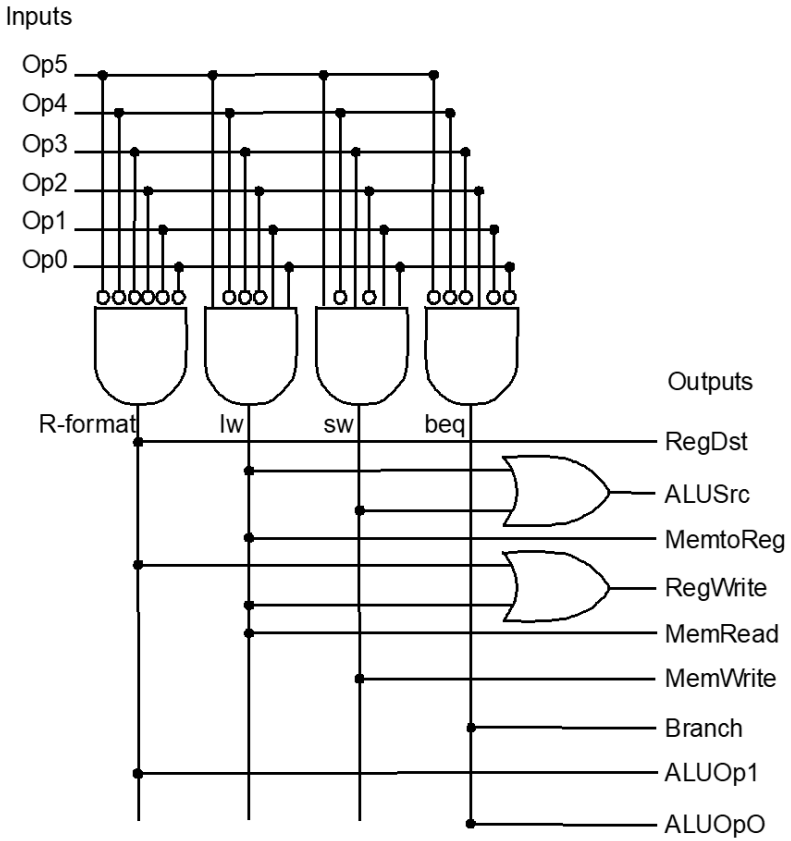
ii. Conflict miss decreases with increasing associativity

iii. Capacity miss does not depend on associativity but decreases with increasing size

Reference Sheet



	Opcode					
	Op5	Op4	Op3	Op2	Op1	Op0
R-type	0	0	0	0	0	0
lw	1	0	0	0	1	1
sw	1	0	1	0	1	1
beq	0	0	0	1	0	0



	RegDst	ALUSrc	MemTo Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop	
								op1	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

	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	ALUop	Instruction Operation	Funct field	ALU action	ALU control
lw	00	load word	xxxxxx	add	0010
sw	00	store word	xxxxxx	add	0010
beq	01	branch equal	xxxxxx	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

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

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

ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

Boolean Algebra Laws		
Identity	$A + 0 = 0 + A = A$	$A \cdot 1 = 1 \cdot A = A$
Complement	$A + A' = 1$	$A \cdot A' = 0$
Commutative	$A + B = B + A$	$A \cdot B = B \cdot A$
Associative	$A + (B + C) = (A + B) + C$	$A \cdot (B \cdot C) = (A \cdot B) \cdot C$
Distributive	$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$	$A + (B \cdot C) = (A + B) \cdot (A + C)$
Idempotency	$X + X = X$	$X \cdot X = X$
One/ Zero Element	$X + 1 = 1$	$X \cdot 0 = 0$
Involution	$(X')' = X$	
Absorption 1	$X + X \cdot Y = X$	$X \cdot (X + Y) = X$
Absorption 2	$X + X' \cdot Y = X + Y$	$X \cdot (X' + Y) = X \cdot Y$
De Morgan's	$(X + Y)' = X' \cdot Y'$	$(X \cdot Y)' = X' + Y'$
Consensus	$X + Y \cdot Z + Y' \cdot Z = X + Y' \cdot Z$	$(X + Y) \cdot (X' + Z) \cdot (Y + Z) = (X + Y) \cdot (X' + Z)$

C Precedence		
Postfix Unary	`++, --`	Left to right
Prefix Unary	`+, -, ++, --, &, *, (type), !`	Right to left
Multiplicative Binary	`*, /, %`	Left to right
Additive Binary	`+, -`	Left to right
Relational	`<=, >=, <, >`	Left to right
Equality	`==, !=`	Left to right
Conjunction	`&&`	Left to right
Disjunction	`||`	Left to right
Assignment	`=, +=, -=, *=, /=, %=`	Right to left

Prefix Multipliers			
Nano	n	10^{-9}	2^{-30}
Micro	µ	10^{-6}	2^{-20}
Milli	m	10^{-3}	2^{-10}
Kilo	k	10^{3}	2^{10}
Mega	M	10^{6}	2^{20}
Giga	G	10^{9}	2^{30}
Tera	T	10^{12}	2^{40}