```
github.com/tangboxuan
```

Number Systems

System	Value	-x	Negation	Addition	Overflow
Sign & Magnitude			Invert first bit		
1s Complement	+ve: same as 2s -ve: 2s + 1	2^n - x - 1	Invert all bits	If MSB carry out: +=1	Result opposite sign of A and B
2s Complement	if x[n-1]: -= 2^(n-1) for i in [n-2..0]: if x[i]: += 2^i	2^n - x	Invert all bits + 1	Ignore MSB carry out	MSB carry in != MSB carry out Result opposite sign of A and B
Excess-n	x-n				
IEEE 754	(Sign) (1.Mantissa) x 2^Exponent		Invert first bit	1: Sign 8: Ex-127 Exp	--- 23: Mantissa

MIPS

Operation	RR1	RR2	WR	WD	Opr1	Opr2	Address	Write Data
lw $a, x($b)	$b	$a	$a	Mem([$b] +x)	[$b]	x	[$b] + x	[$a]
beq $a,$b, immd	$a	$b	$b	?	[$a]	[$b]	[$a] - [$b]	[$b]
sub $a, $b, $c	$b	$c	$a	[$b] – [$c]	[$b]	[$c]	[$b] – [$c]	[$c]
addi $a, $b, immd	$b	$a	$a	[$b] + immd	[$b]	immd	[$b] + immd	[$a]

MIPS Reference Data

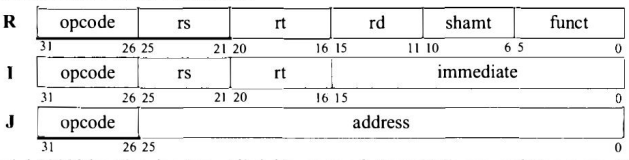
①

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	0 / 20_{hex}
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		0 / 21_{hex}
And	and	R	R[rd] = R[rs] & R[rt]		0 / 24_{hex}
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	jr	R	PC=R[rs]		0 / 08_{hex}
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs]+SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs]+SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])		0 / 27_{hex}
Or	or	R	R[rd] = R[rs] | R[rt]		0 / 25_{hex}
Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		0 / $2a_{hex}$
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0	(2)	a_{hex}
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b_{hex}
Set Less Than Unsig.	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	0 / $2b_{hex}$
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt		0 / 00_{hex}
Shift Right Logical	srl	R	R[rd] = R[rt] >> shamt		0 / 02_{hex}
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28_{hex}
Store Conditional	sc	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0	(2,7)	38_{hex}
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29_{hex}
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	0 / 22_{hex}
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		0 / 23_{hex}

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

BASIC INSTRUCTION FORMATS

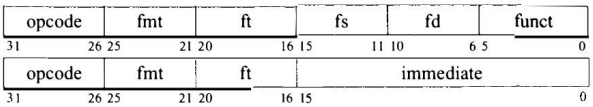


ARITHMETIC CORE INSTRUCTION SET

② OPCODE / FMT /FT / FUNCT (Hex)

NAME, MNEMONIC		FOR-MAT	OPERATION		
Branch On FP True	bc1t	FI	if(FPcond)PC=PC+4+BranchAddr	(4)	11/8/1/--
Branch On FP False	bc1f	FI	if(!FPcond)PC=PC+4+BranchAddr	(4)	11/8/0/--
Divide	div	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]		0/--/--/1a
Divide Unsigned	divu	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	(6)	0/--/--/1b
FP Add Single	add.s	FR	F[fd]= F[fs] + F[ft]		11/10/--/0
FP Add Double	add.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}		11/11/--/0
FP Compare Single	c.x.s*	FR	FPcond = (F[fs] *op* F[ft]) ? 1 : 0		11/10/--/*y*
FP Compare Double	c.x.d*	FR	FPcond = ({F[fs],F[fs+1]} *op* {F[ft],F[ft+1]}) ? 1 : 0		11/11/--/*y*
FP Divide Single	div.s	FR	F[fd] = F[fs] / F[ft]		11/10/--/3
FP Divide Double	div.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}		11/11/--/3
FP Multiply Single	mul.s	FR	F[fd] = F[fs] * F[ft]		11/10/--/2
FP Multiply Double	mul.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}		11/11/--/2
FP Subtract Single	sub.s	FR	F[fd]=F[fs] - F[ft]		11/10/--/1
FP Subtract Double	sub.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}		11/11/--/1
Load FP Single	lwc1	I	F[rt]=M[R[rs]+SignExtImm]	(2)	31/--/--/--
Load FP Double	ldc1	I	F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4]	(2)	35/--/--/--
Move From Hi	mfhi	R	R[rd] = Hi		0 /--/--/10
Move From Lo	mflo	R	R[rd] = Lo		0 /--/--/12
Move From Control	mfc0	R	R[rd] = CR[rs]		10 /0/--/0
Multiply	mult	R	{Hi,Lo} = R[rs] * R[rt]		0/--/--/18
Multiply Unsigned	multu	R	{Hi,Lo} = R[rs] * R[rt]	(6)	0/--/--/19
Shift Right Arith.	sra	R	R[rd] = R[rt] >>> shamt		0/--/--/3
Store FP Single	swc1	I	M[R[rs]+SignExtImm] = F[rt]	(2)	39/--/--/--
Store FP Double	sdc1	I	M[R[rs]+SignExtImm+4] = F[rt+1]	(2)	3d/--/--/--

FLOATING-POINT INSTRUCTION FORMATS



PSEUDOINSTRUCTION SET

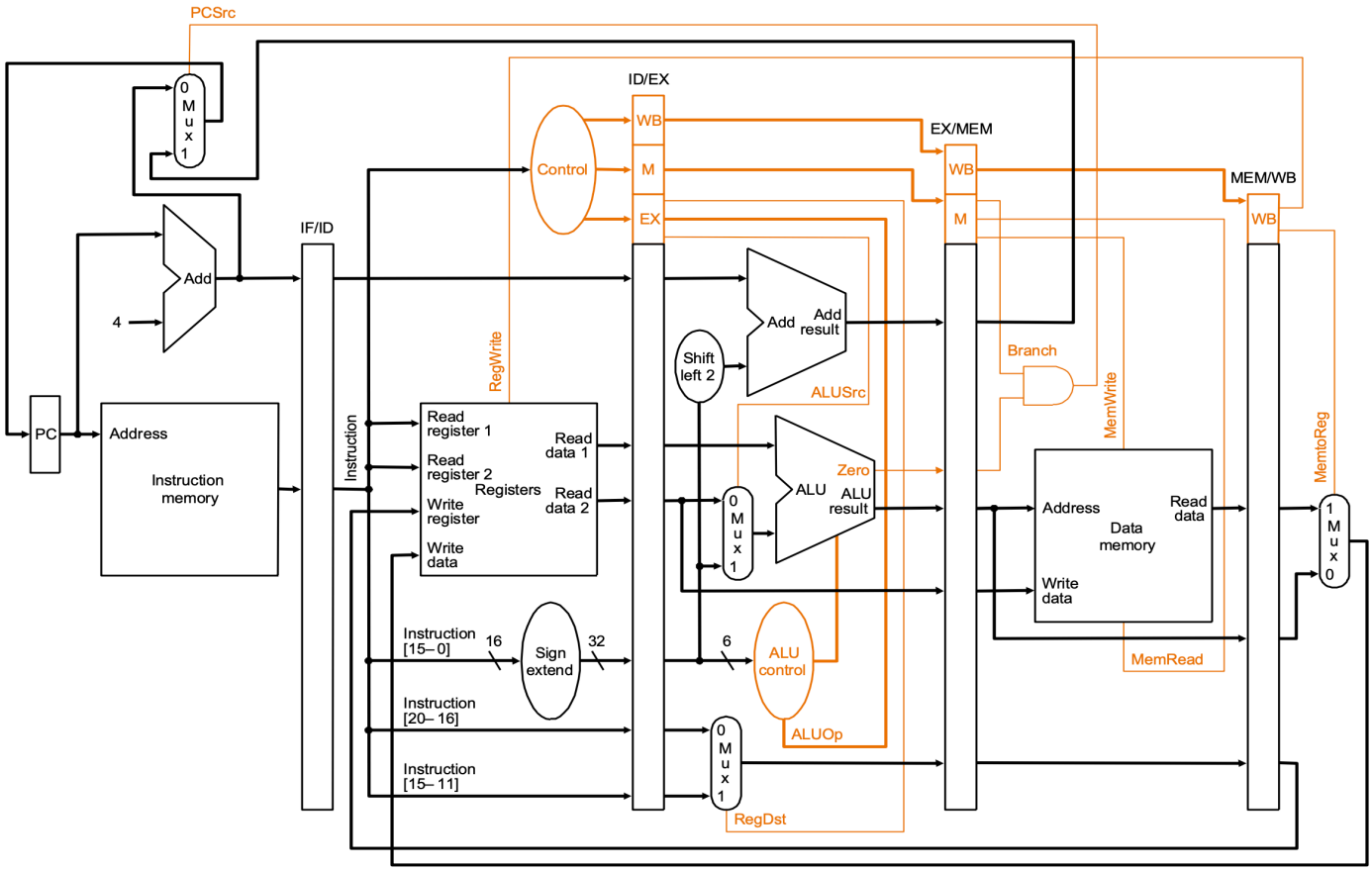
NAME	MNEMONIC	OPERATION
Branch Less Than	blt	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	bgt	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	ble	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	bge	if(R[rs]>=R[rt]) PC = Label
Load Immediate	li	R[rd] = immediate
Move	move	R[rd] = R[rs]

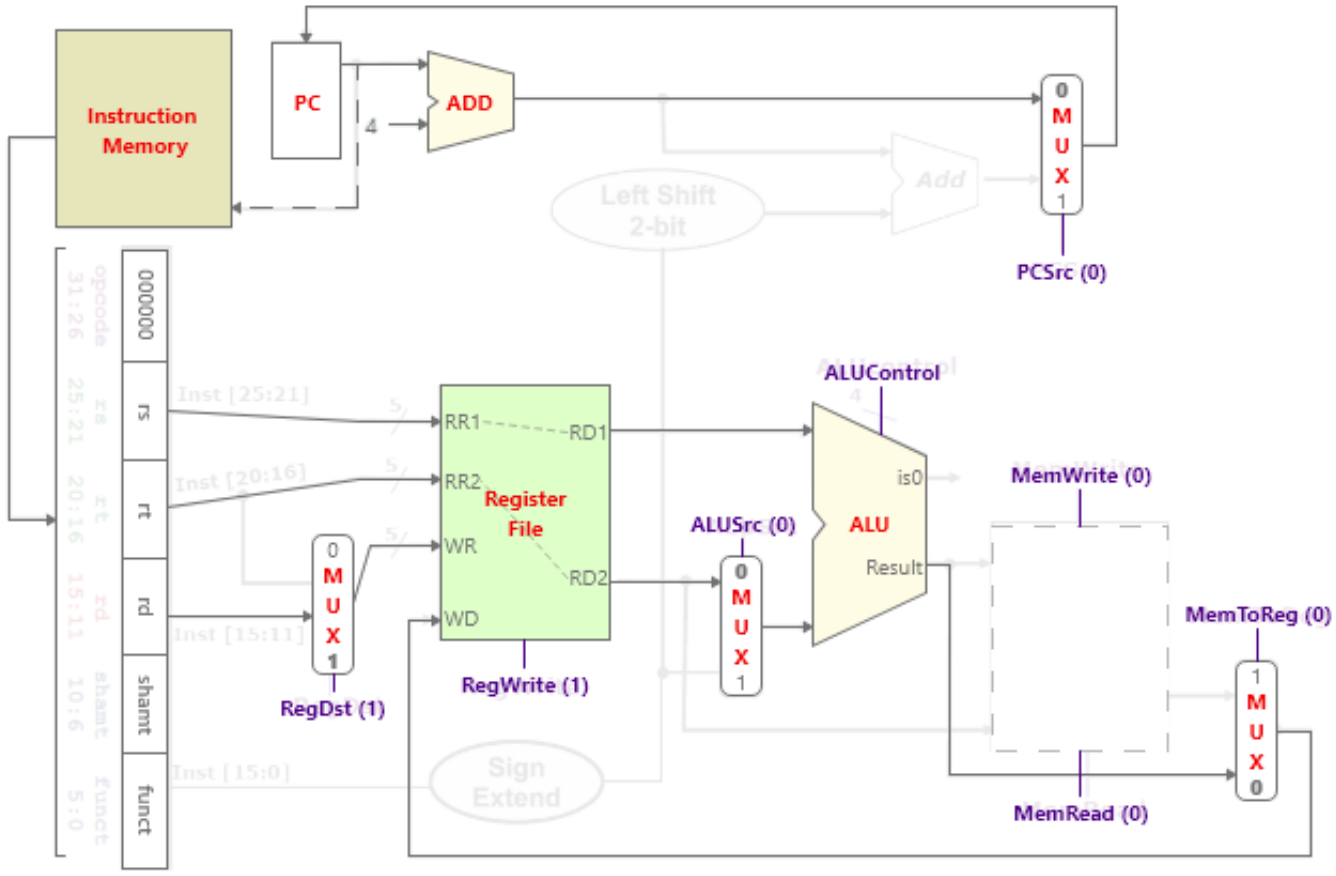
REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$vl	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes

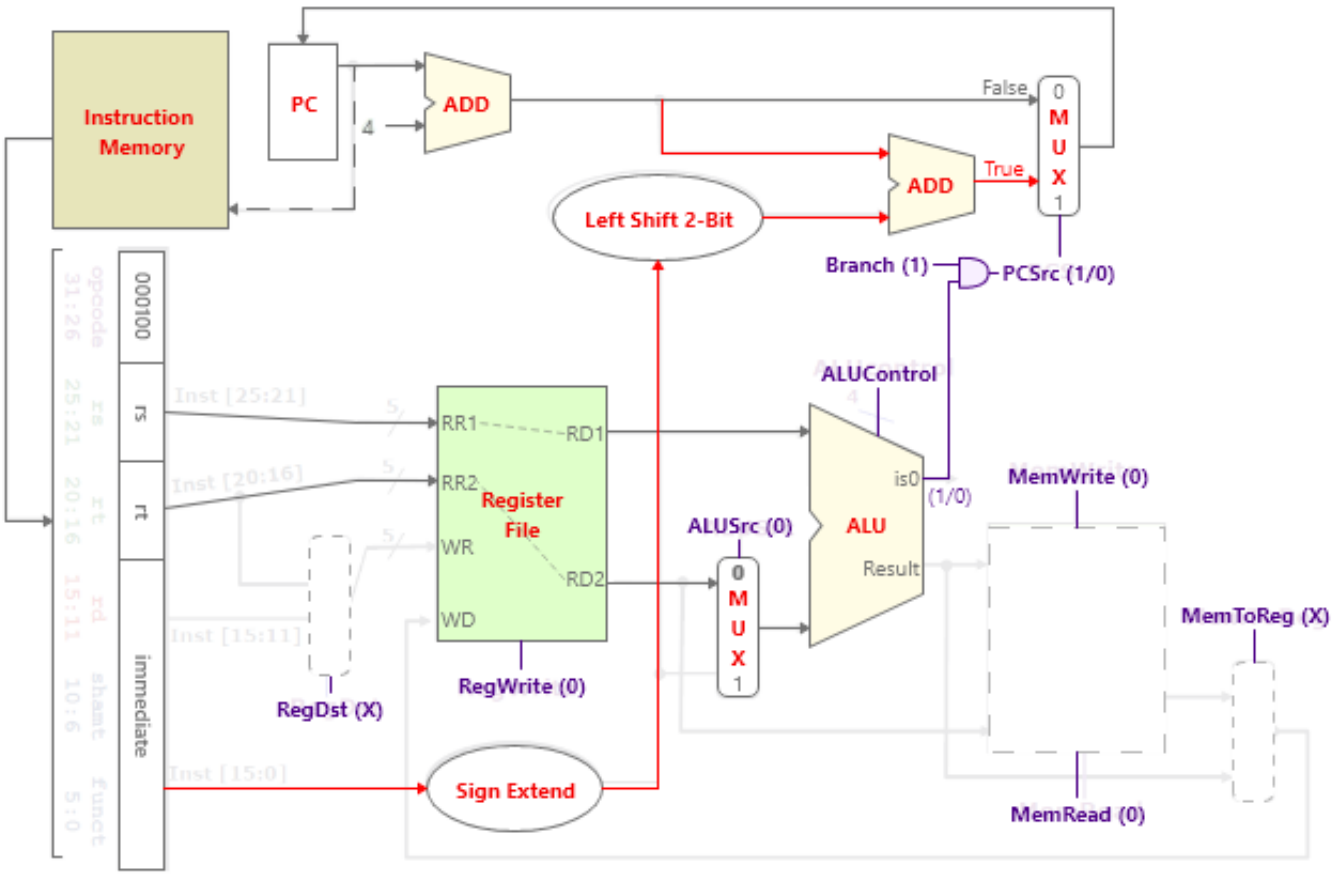
	EX Stage				MEM Stage			WB Stage	
	RegDst	ALUSrc	ALUop		Mem Read	Mem Write	Branch	MemTo Reg	Reg Write
			op1	op0					
R-type	1	0	1	0	0	0	0	0	1
lw	0	1	0	0	1	0	0	1	1
sw	X	1	0	0	0	1	0	X	0
beq	X	0	0	1	0	0	1	X	0

	Opcode (Op[5:0] == Inst[31:26])						
	Op5	Op4	Op3	Op2	Op1	Op0	Value in Hexadecimal
R-type	0	0	0	0	0	0	0
lw	1	0	0	0	1	1	23
sw	1	0	1	0	1	1	2B
beq	0	0	0	1	0	0	4

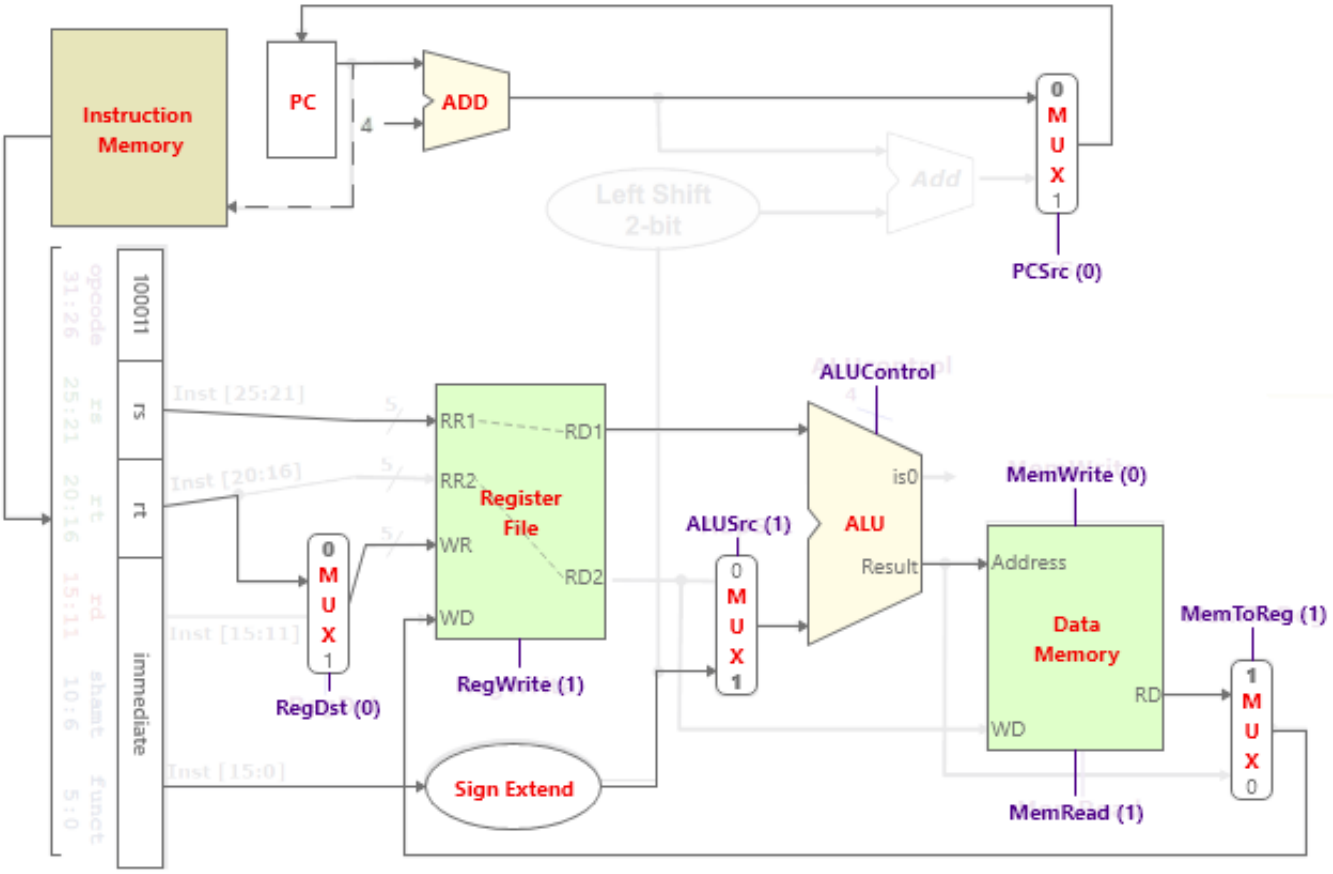



Instruction Memory
PC
4
ADD
0 M U X 1
PCSrc (0)
Left Shift 2-bit
Add
000000
rs
rt
rd
shamt
funct
Inst [25:21]
Inst [20:16]
Inst [15:11]
Inst [15:0]
RR1
RD1
RR2
Register File
WR
RD2
WD
RegWrite (1)
RegDst (1)
ALUControl
ALUSrc (0)
ALU
is0
Result
MemWrite (0)
MemRead (0)
MemToReg (0)
Sign Extend

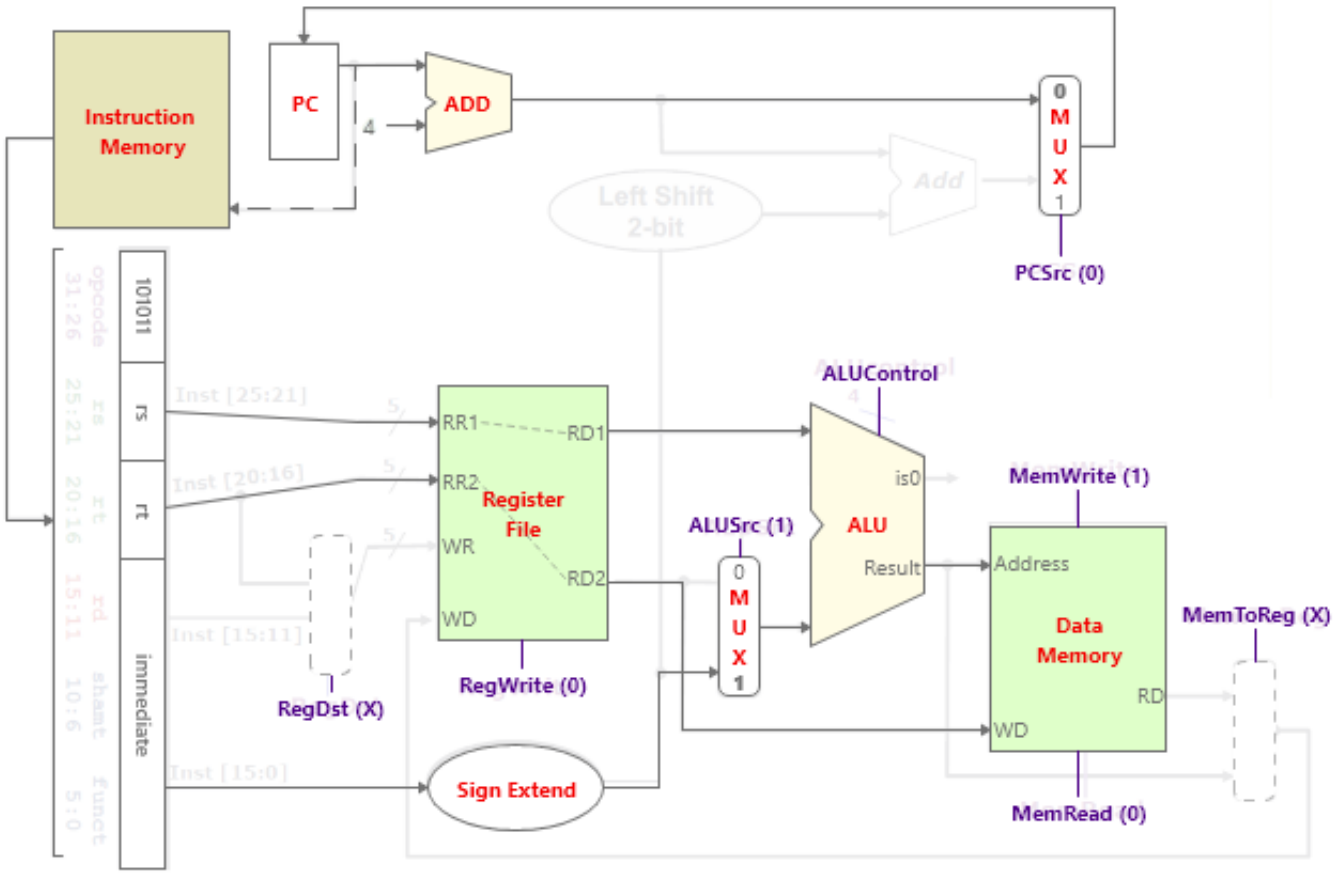
Add


Instruction Memory
PC
4
ADD
False
True
Left Shift 2-Bit
Branch (1)
PCSrc (1/0)
000100
rs
rt
immediate
Inst [25:21]
Inst [20:16]
Inst [15:11]
Inst [15:0]
RR1
RD1
RR2
Register File
WR
RD2
WD
RegWrite (0)
RegDst (X)
ALUControl
ALUSrc (0)
ALU
is0
(1/0)
Result
MemWrite (0)
MemRead (0)
MemToReg (X)
Sign Extend

BEQ



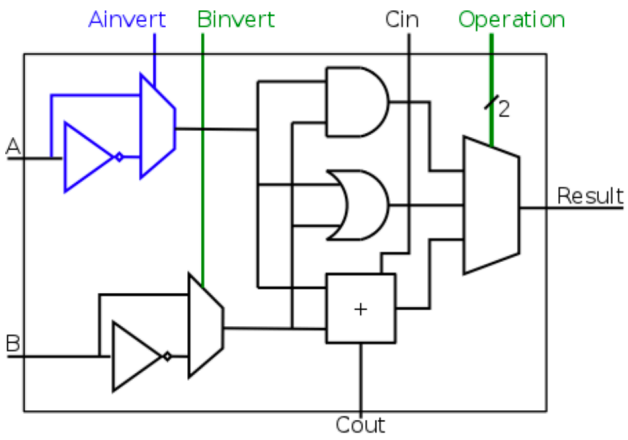
Load Word



Store Word

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

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



Dependency	Solution	Inst	Stall
Data	None	add	2
Data	None	lw	2
Data	Fwd	add	0
Data	Fwd	lw	1
Control	None	ALL	3

Dependency	Solution	Inst	Stall
Control	Early		1
Control	Early	add	2
Control	Early	lw	3
Control	Predict	YES	0
Comtrol	Delay		0

Boolean Algebra

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)

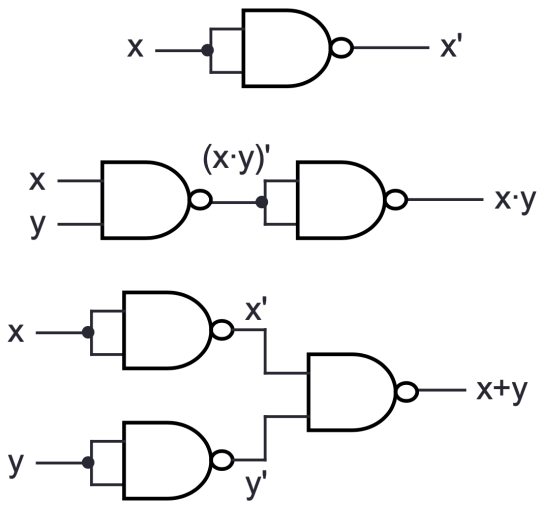
Idempotency	
X + X = X	X · X = X
One element / Zero element	
X + 1 = 1	X · 0 = 0
Involution	
(X')' = X	
Absorption 1	
X + X·Y = X	X·(X + Y) = X
Absorption 2	
X + X'·Y = X + Y	X·(X' + Y) = X·Y
DeMorgans' (can be generalised to more than 2 variables)	
(X + Y)' = X' · Y'	(X · Y)' = X' + Y'
Consensus	
X·Y + X'·Z + Y·Z = X·Y + X'·Z	(X+Y)·(X'+Z)·(Y+Z) = (X+Y)·(X'+Z)

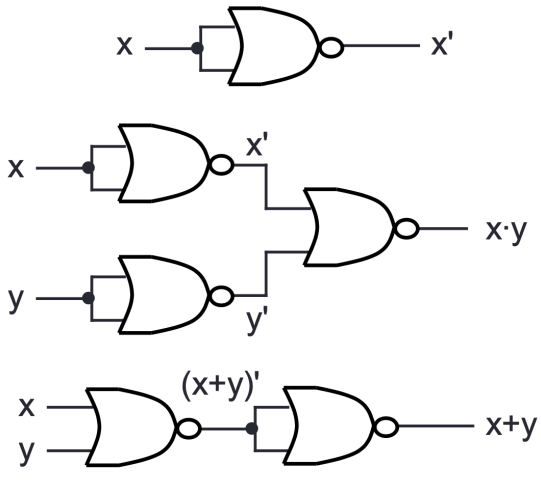
x	y	Minterms		Maxterms	
		Term	Notation	Term	Notation
0	0	x'·y'	m0	x+y	M0
0	1	x'·y	m1	x+y'	M1
1	0	x·y'	m2	x'+y	M2
1	1	x·y	m3	x'+y'	M3

Duality: If the AND/OR operators and 0/1 identity elements are interchanged, it remainds valid

x+1 = 1 -> x.0 = 0

Logic Circuits



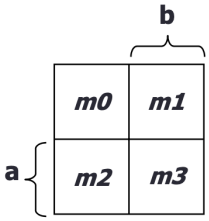


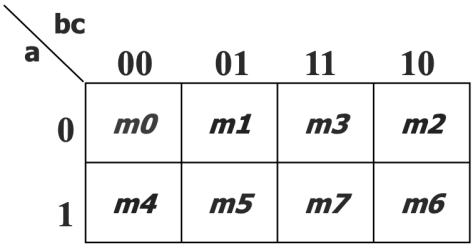
SOP

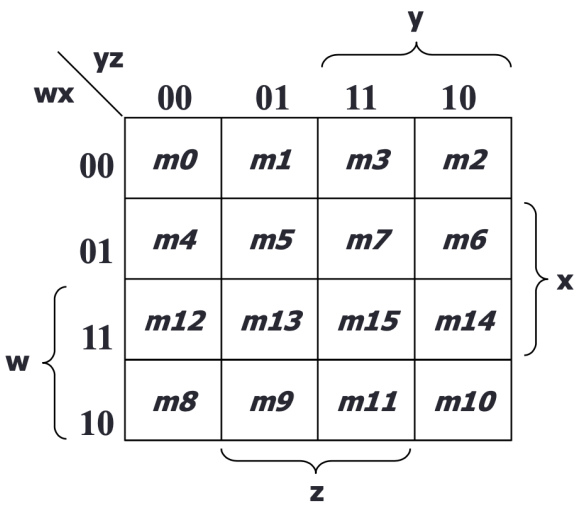
1. AND-OR
2. NAND-NAND

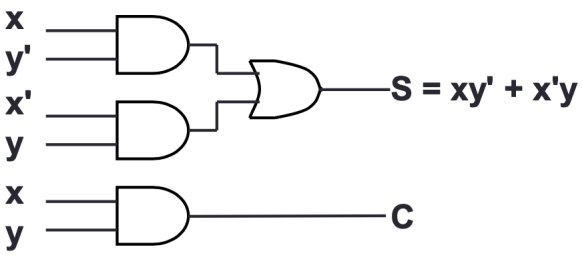
POS

1. OR-AND
2. NOR-NOR

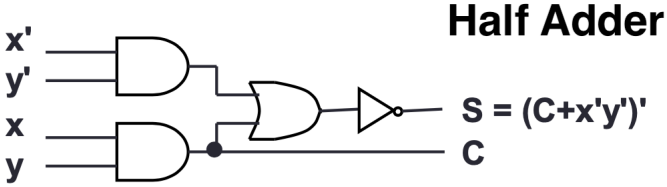


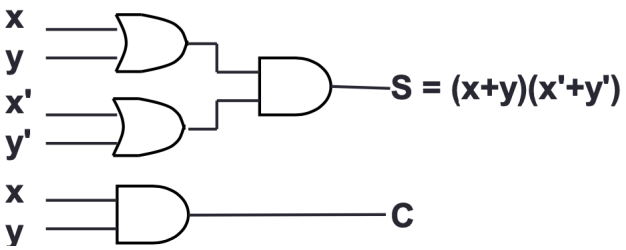


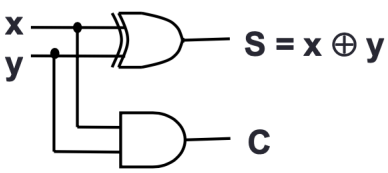




Half Adder

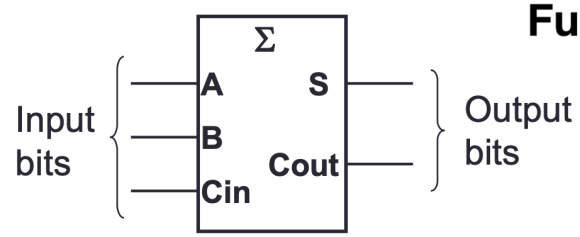


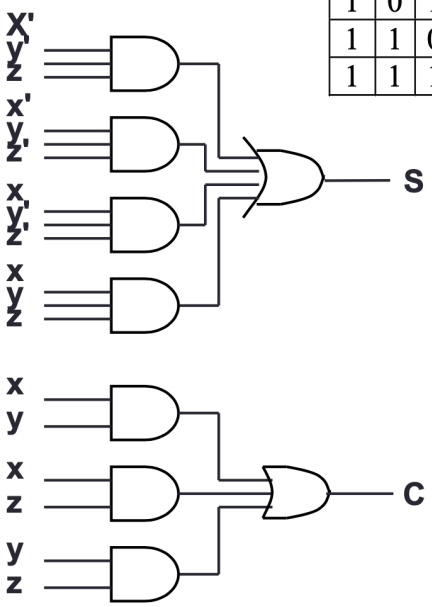


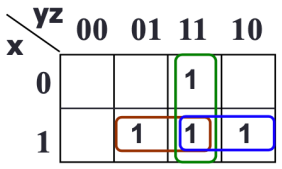


Full Adder

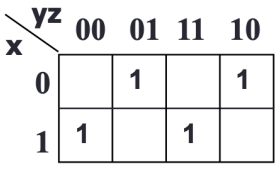
x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



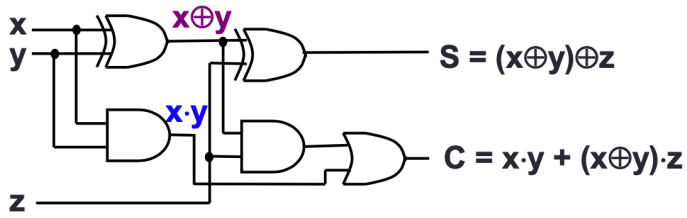




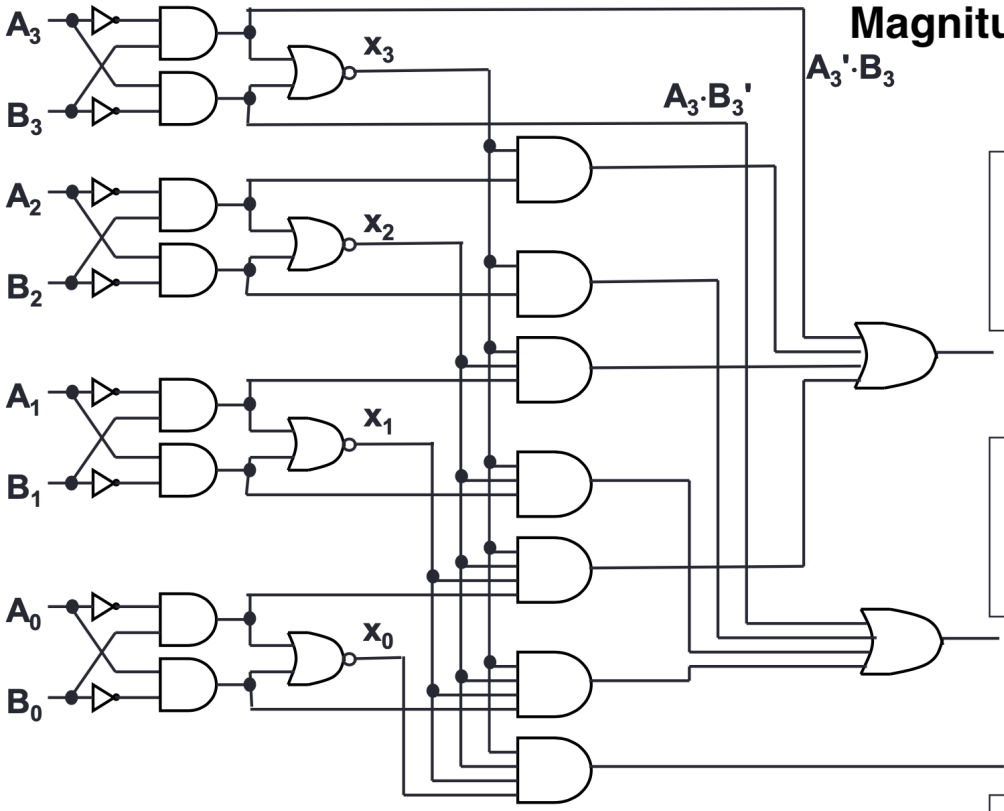
C = xy + xz + yz



S = x'y'z + x'yz' + xy'z' + xyz



Magnitude Comparator



$A_3'\cdot B_3 + x_3\cdot A_2'\cdot B_2 + x_3\cdot x_2.A_1'\cdot B_1 + x_3\cdot x_2\cdot x_1\cdot A_0'\cdot B_0$ (A < B)

$A_3\cdot B_3' + x_3\cdot A_2\cdot B_2' + x_3\cdot x_2\cdot A_1\cdot B_1' + x_3\cdot x_2\cdot x_1\cdot A_0\cdot B_0'$ (A > B)

(A = B)

$x_3\cdot x_2\cdot x_1\cdot x_0$

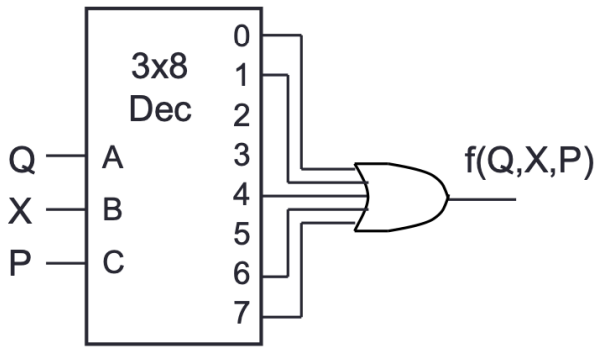
Let $A = A_3A_2A_1A_0$, $B = B_3B_2B_1B_0$; $x_i = A_i\cdot B_i + A_i'\cdot B_i'$

Functions using MSI

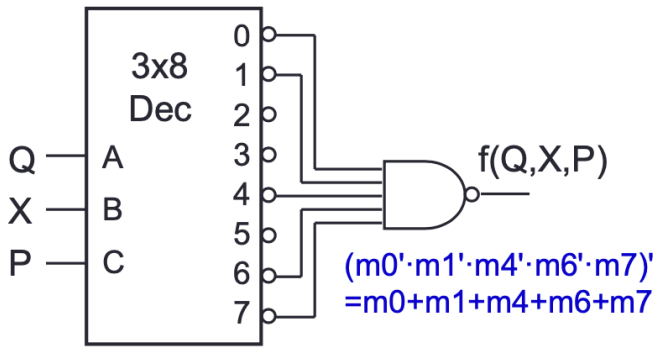
Demultiplexers are similar to a decoder (select) with enable (data)

Multiplexers are similar to a decoder (select) added with 2^n input lines (input)

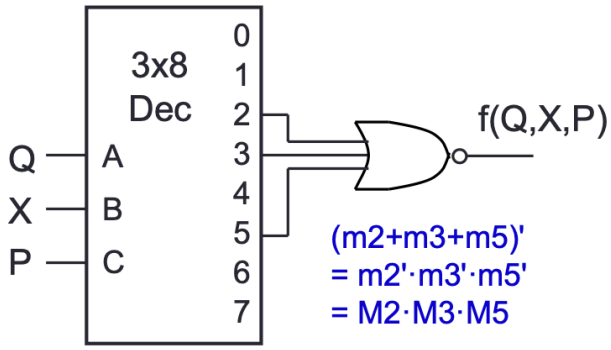
$$f(Q,X,P) = \Sigma m(0,1,4,6,7) = \prod M(2,3,5)$$



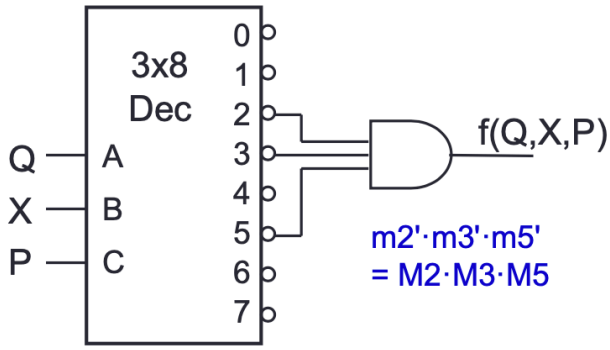
(a) Active-high decoder with OR gate.



(b) Active-low decoder with NAND gate.

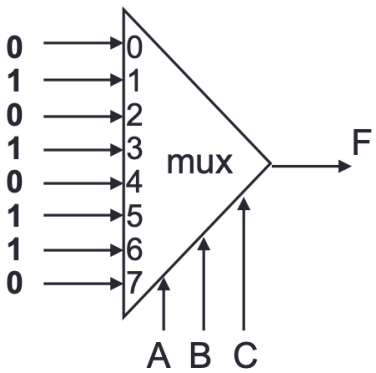


(c) Active-high decoder with NOR gate.



(d) Active-low decoder with AND gate.

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



This method works because:

$$\text{Output} = I_0 \cdot m_0 + I_1 \cdot m_1 + I_2 \cdot m_2 + I_3 \cdot m_3 + I_4 \cdot m_4 + I_5 \cdot m_5 + I_6 \cdot m_6 + I_7 \cdot m_7$$

Supplying '1' to I_1, I_3, I_5, I_6 , and '0' to the rest:

$$\text{Output} = m_1 + m_3 + m_5 + m_6$$

Sequential Logic

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

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

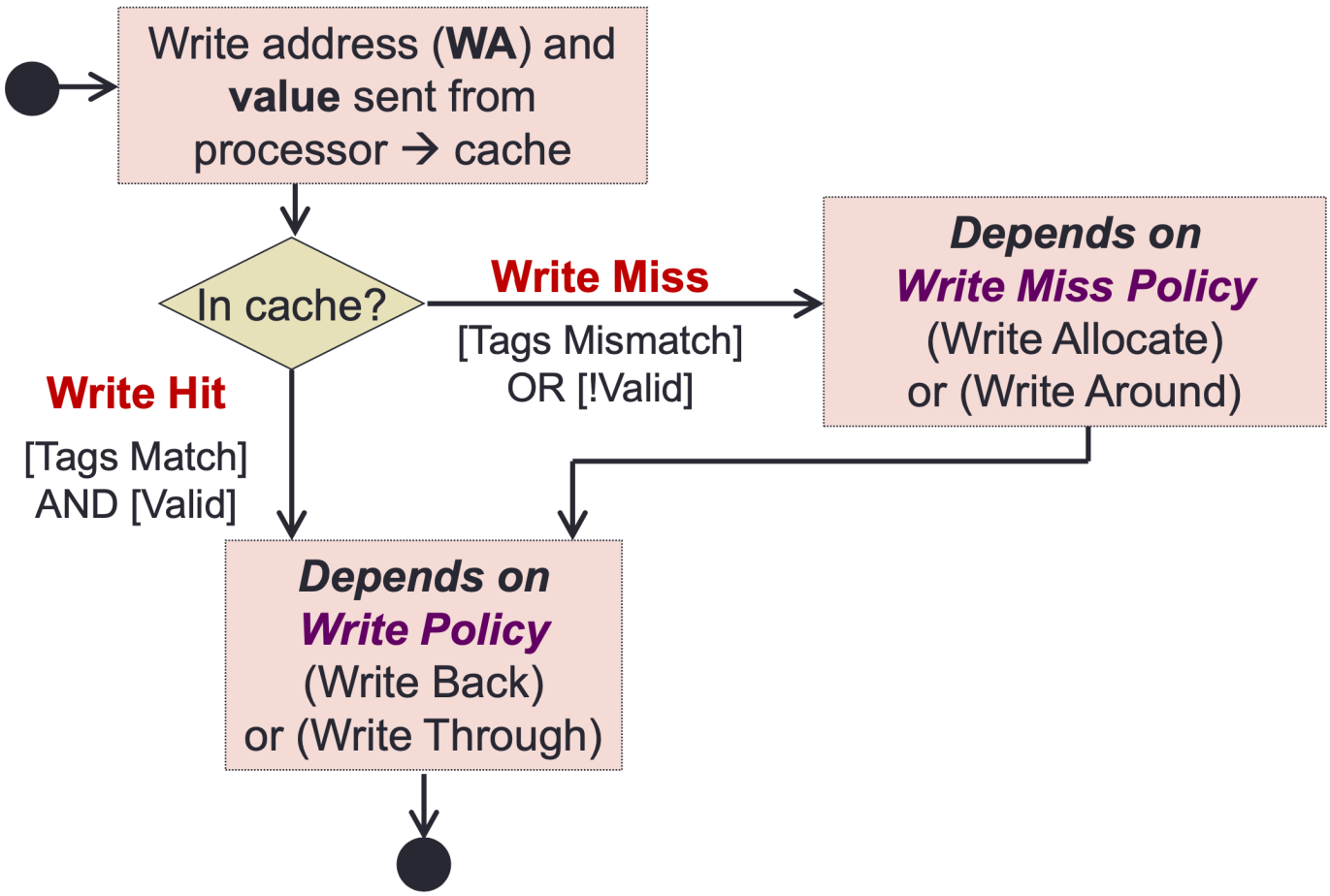
Q	Q^+	D
0	0	0
0	1	1
1	0	0
1	1	1

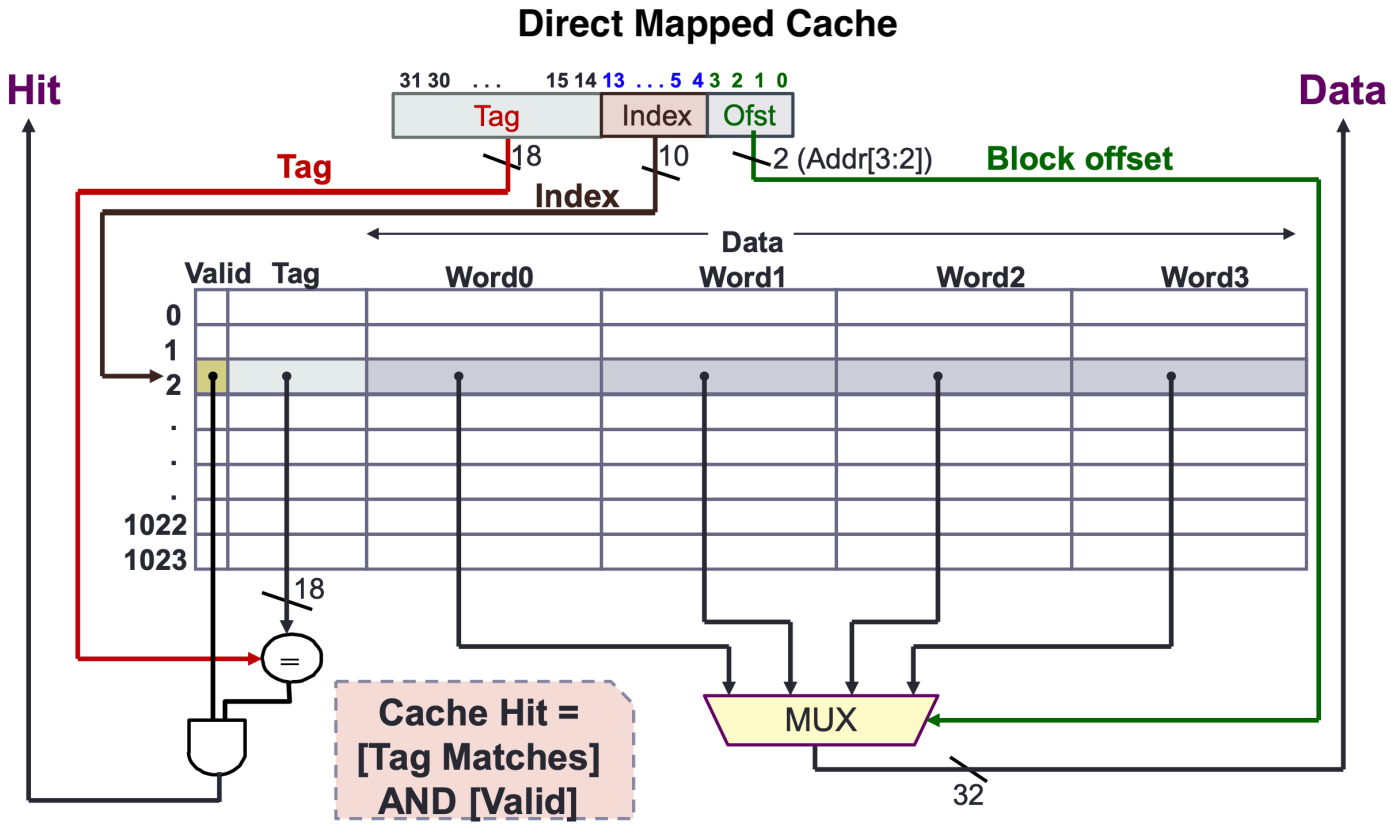
D Flip-flop

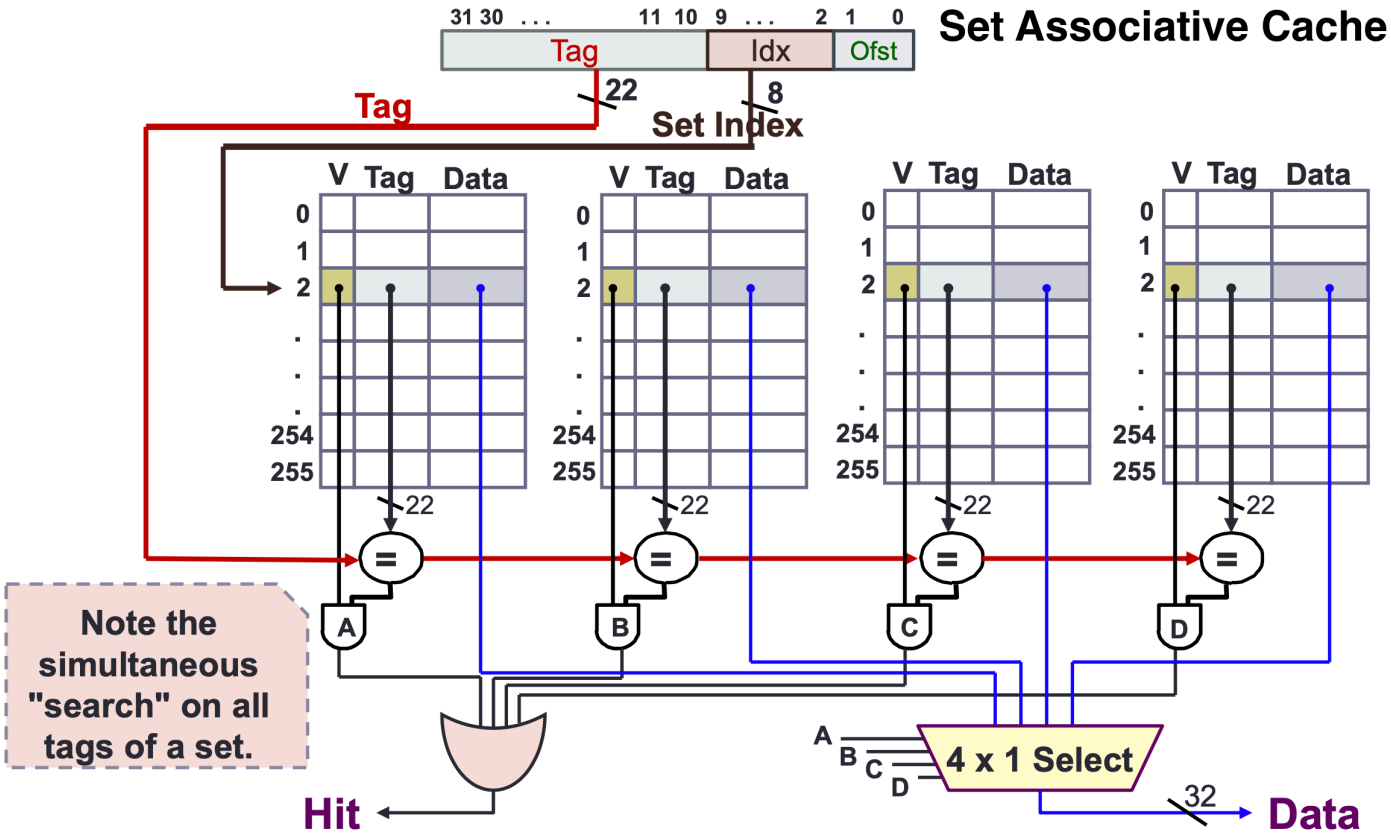
Q	Q^+	T
0	0	0
0	1	1
1	0	1
1	1	0

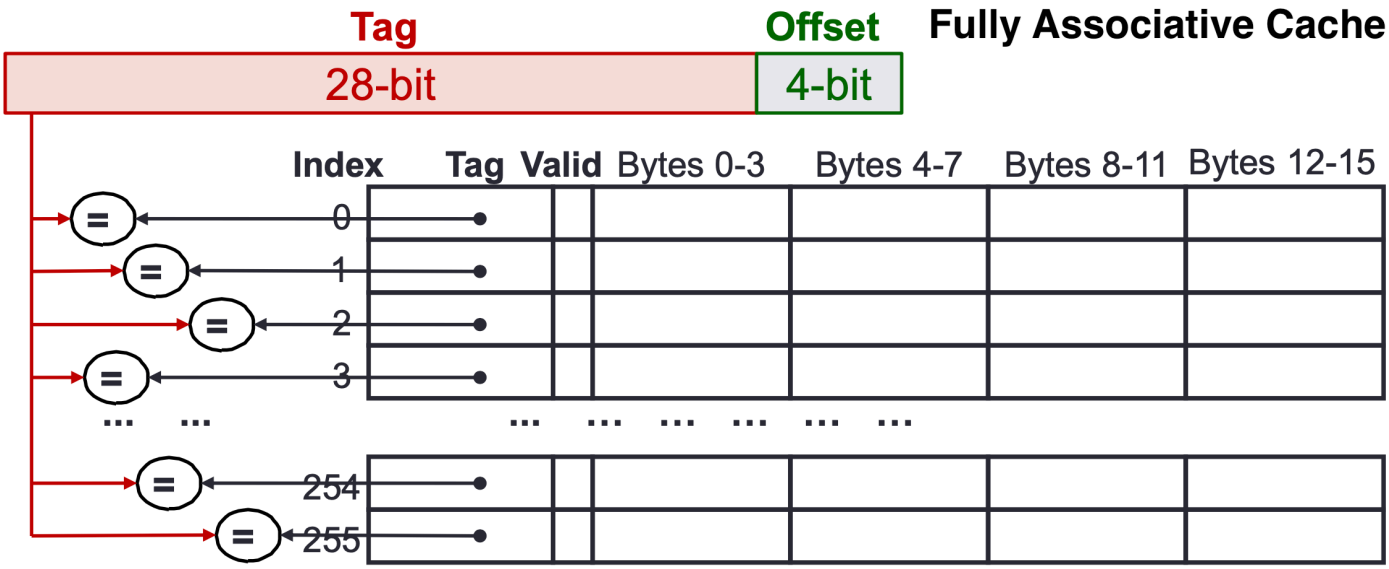
T Flip-flop

Cache


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


Direct Mapped Cache
Hit
Data
31 30 . . . 15 14 13 . . . 5 4 3 2 1 0
Tag
Index
Ofst
18
10
2 (Addr[3:2])
Block offset
Tag
Index
Data
Valid
Tag
Word0
Word1
Word2
Word3
0
1
2
1022
1023
18
=
Cache Hit = [Tag Matches] AND [Valid]
MUX
32





Miss	Description
Compulsory	First access to a block
Conflict	Block has been replaced in cache
Capacity	Cache cannot contain all blocks

Replacement Policy
Least Recently Used
First in First Out
Random Replacement
Least Frequently Used