1. Number systems

① Conversion between bases

1) whole numbers: subtraction of Base^n

2) fraction: repeated multiplication until fractional product is 0 to sufficient dp

eg. $0.3125_{10} = 0.0101_2$

$0.3125 \times 2 = 0.625$

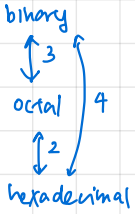
$0.625 \times 2 = 1.25$

$0.25 \times 2 = 0.50$

$0.50 \times 2 = 1.00$

3) convert bases

Hexadecimal	Binary	Decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15



4) range and overflow

N bits – 2^N values

max $\overset{+1}{\rightleftharpoons}$ min (−1)

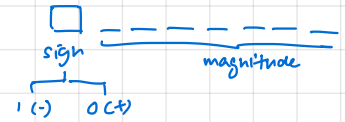
5) binary addition and subtraction

- $A - B = A + (-B)$ ← negate
- detecting overflows: positive (0) + positive (0) → negative (1)

  negative (1) + negative (1) → positive (0)

② representing negative numbers

sign & magnitude



- range: $\pm(2^N-1)$

$-2^N+1 \ldots -1, -0, 0, 1 \ldots 2^N-1$

- to negate: flip sign

1's complement

$-(2^7-1)$

$2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 1$

- range: $\pm(2^N-1)$

0111 1111 127 largest

1000 000 -127 smallest

0000 0000 +0

1111 1111 -0

- to negate: flip all bits
- how to obtain:

$-x = 2^n - x - 1$ (binary)

$-x = B^n - B^m - x$ (general) } convert to binary

n: no. of integer digits; m: no. of fractional digits

- binary addition: if end carry, add back to result

excess representation

- range: -2^{N-1} to $2^{N-1}-1$
- convert binary to decimal − excess (excess $= 2^{N-1}$)

2's complement

-2^7

$2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 1$

- range: -2^{N-1} to $2^{N-1}-1$

0 111 1111 127 max

1 000 000 -128 min

0000 0000 0

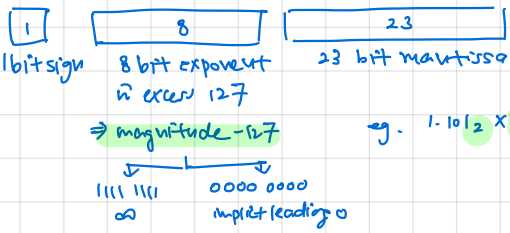
- to negate: invert all, binary addition +1
- how to obtain:

$-x = 2^N - x$

$-x = B^N - x$ } convert to binary

- addition: binary addition, discard end carry

③ IEEE 754 floating point representation

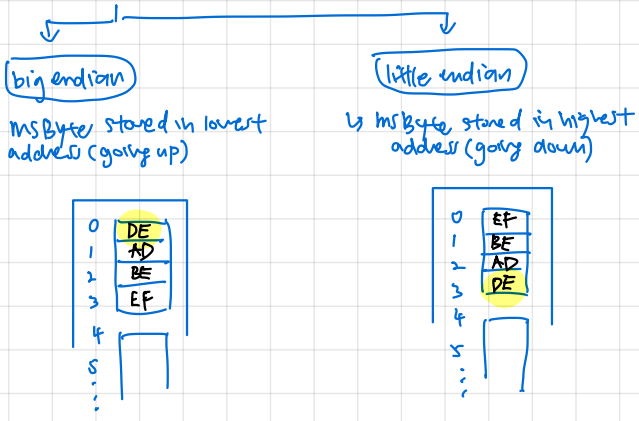


eg. $1.101_2 \times 2^4$

4. ISAs

1) address space: N bit address → 2^N space

2) endianness



3) instruction set size

- more bits = more instructions
- use single unique opcodes to distinguish } max / min
- "stage-wise" splitting

2. Assembly & MIPS

① useful logical operations & masking

sll , srl , and , andi , or , ori , nor , xor

④ ⇒ ~(or( ))

- sll → multiply divide by 2^n, removes value / sign extension as bits shifted
- and, andi, or, ori, nor, xor → sequential bitwise comparison

useful notes:

and	or	xor	nor
x and 0 = 0	x or 1 = 1	x xor x = 0	x nor 1 = 0
x and 1 = x	x or 0 = x	x xor 0 = x	x nor x = flip
x and x = 1		x xor 1 = flipped	x nor 0 = flip
		x xor flipped = 1	
		associative → stacks to swap	

- copy to temp: andi temp , $1 , 00100001
- clear target positions: andi $2 $2 , 1110 0011
- paste: or $2 $2 , temp
- flip and copy to temp:
  xor temp , $1 , 0001 0001 # flip
  andi temp , temp , 0001 0001
  # hide positions of non interest

---

3. C programming

① computation

1) variable assignment : assignment returns to LHS , cascading

Operator Type	Operator	Associativity
Primary expression operators	( ) expr++ expr--	Left to right
Unary operators	* & + - ++expr --expr (typecast)	Right to left
Binary operators	* / %	Left to right
	+ -	
Assignment operators	= += -= *= /= %=	Right to left

value updates ← — op — →

2) mixed type operations & truncating

```
int   m = 10/4;      means   m = 2;      int = int/int
float p = 10/4;      means   p = 2.0;    float = int/int
int   n = 10/4.0;    means   n = 2;      int = int/float
float q = 10/4.0;    means   q = 2.5;    float = int/float
int   r = -10/4.0;   means   r = -2;     Caution! int/float
```

3) types

- int : 4 bytes
- float : 4 byte
- double : 8 byte
- char : single quotes , 1 byte
- no boolean types (0 or 1)

② pointers

1) declaration and initialisation

```
type * pointer_name ;          // points to random address if not initialised
type * pointer_name = &a ;
```

2) changing addr. : pointer_name = &a;

3) dereferencing : * pointer_name = value;

③ functions and variable scope

1) scope : local functions are local

④ arrays

1) continuous memory location
2) cannot initialise after declaration → random if not at all initialised; 0 for non-initialised if partially
3) a is fixed pointer to a[0]
4) sizeof( ) → expression → size in bytes of object representation of return value.
   → type → size in bytes of type
   → pointer to arr. into function → size of pointer
5) array sizes depend on type — int, double etc. = 4; char[] = String = 1

⑤ strings

1) input & output , functions → see notes
2) importance of '\0' ⇒ distinguishes String from char[]

⑥ structures

1) reading and modifying : struct_variable . field
2) structure variable pointer points to first field in struct
3) pointer is mutable
4) pass by value by default
5) arrow operator

(* struct_pointer).field ⇔ struct_pointer → field ⇔ *(struct_pointer.field)

equivalent, . precedence over *

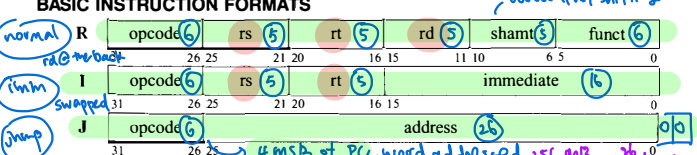
MIPS Reference Data

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

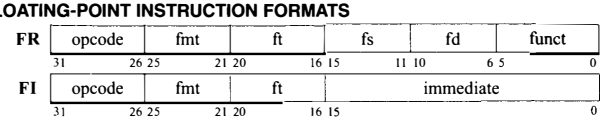


ARITHMETIC CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION		OPCODE / FMT /FT / FUNCT (Hex)
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/--
Store FP Double	sdc1	I	M[R[rs]+SignExtImm] = F[rt]; M[R[rs]+SignExtImm+4] = F[rt+1]	(2)	3d/--/--/--

Hexadecimal	Binary	Decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

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]

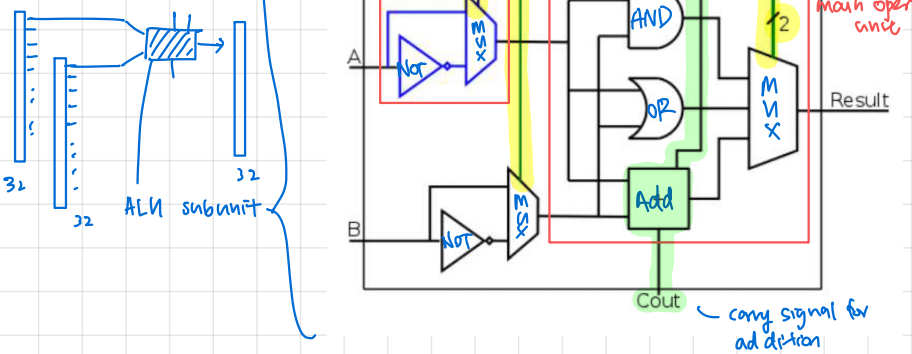
REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	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

6. control

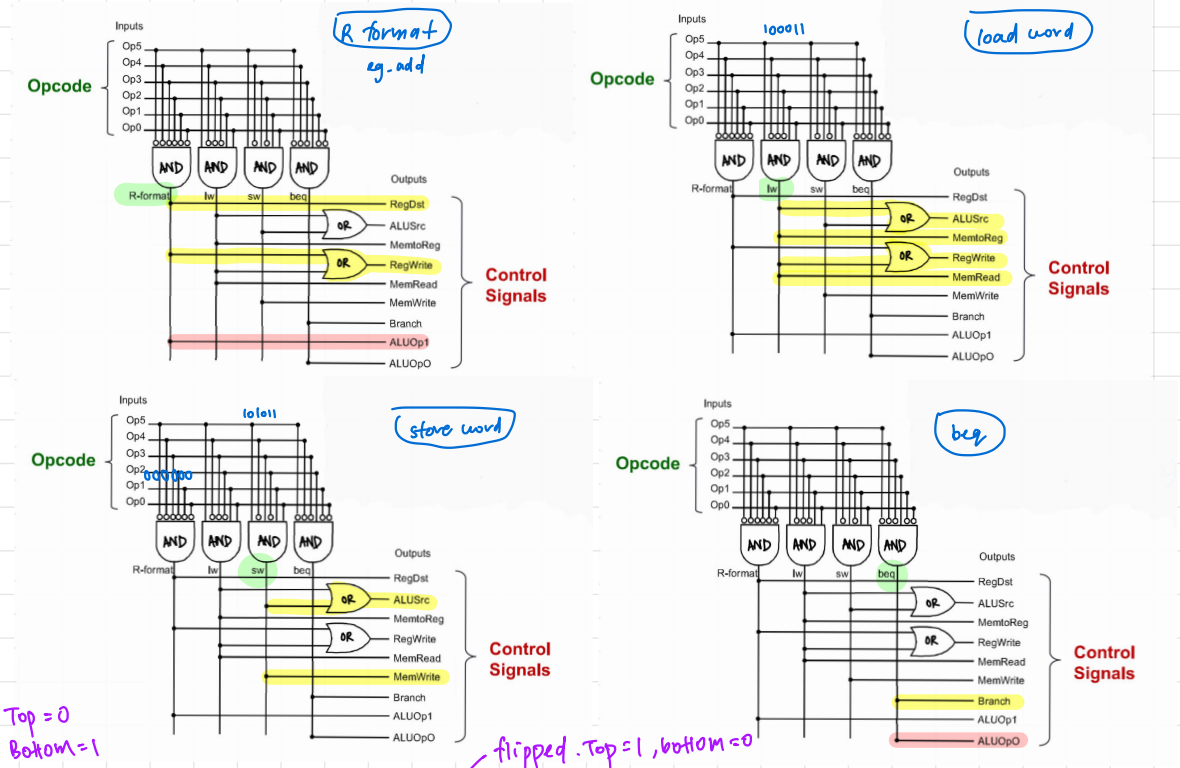
① ALU implementation

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR



② combinational circuits for signals

1) generating non-ALU signals



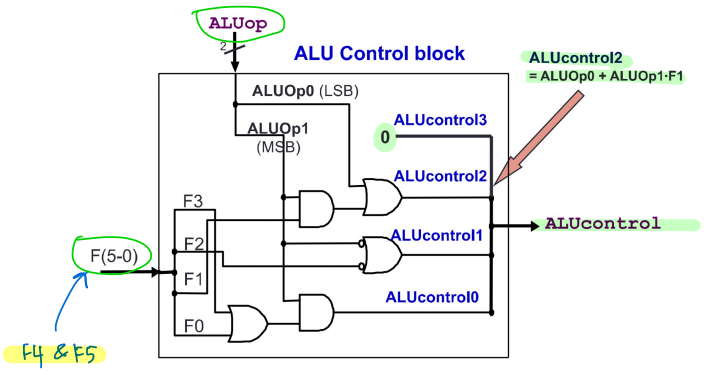
Top = 0
Bottom = 1

flipped. Top = 1, bottom = 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
I-type	0	1	0	1	0	0	0	1	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
I-type	0	0	0	0	0	0	0

2) generating ALUctrl signals



F4 & F5 are not used at all for the standard instruction set (simplified diagram)

	ALUop		Funct Field (F[5:0] == Inst[5:0])						ALU control
	MSB	LSB	F5	F4	F3	F2	F1	F0	
lw	0	0	X	X	X	X	X	X	0 0 1 0
sw	0	0	X	X	X	X	X	X	0 0 1 0
beq	~~0~~ X	1	X	X	X	X	X	X	0 1 1 0
add	1	~~0~~ X	~~1~~ X	~~0~~ X	0	0	0	0	0 0 1 0
sub	1	~~0~~ X	~~1~~ X	~~0~~ X	0	0	1	0	0 1 1 0
and	1	~~0~~ X	~~1~~ X	~~0~~ X	0	1	0	0	0 0 0 0
or	1	~~0~~ X	~~1~~ X	~~0~~ X	0	1	0	1	0 0 0 1
slt	1	~~0~~ X	~~1~~ X	~~0~~ X	1	0	1	0	0 1 1 1

add
sub
add
sub
and
or
slt

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

---

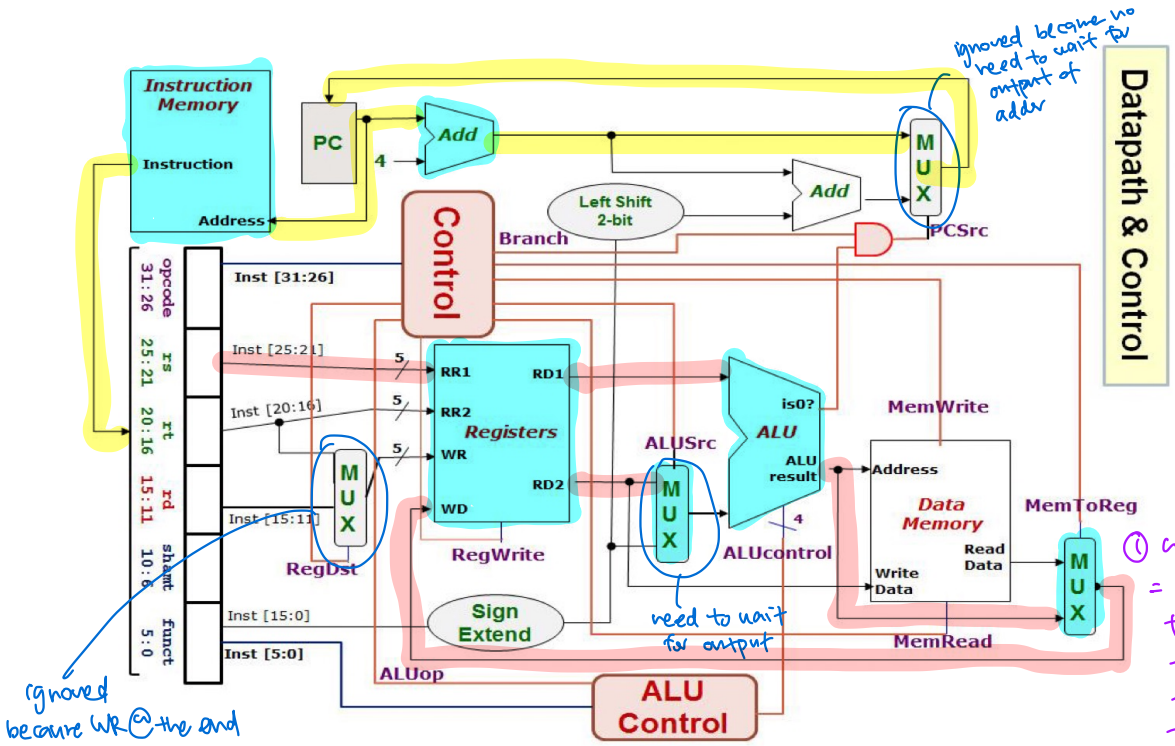
3) cycle time

- single cycle implementation : T = time of longest possible process
- multi cycle implementation : T = time of longest single stage

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

arithmetic op rd rs rt

② others : instruct + register + mux (regDst)
instruct + control ... only diff control reg
instruction + ADD + MUX (PCSrc)



ignored because no need to wait for output of adder

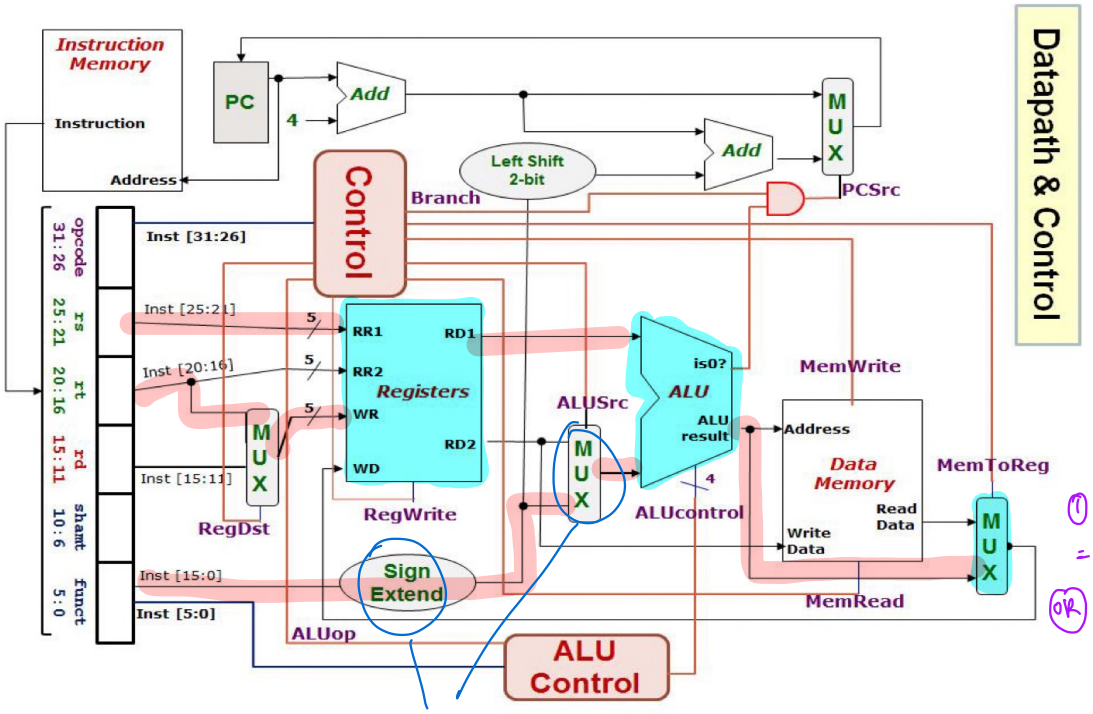
ignored because WR @ the end

need to wait for output

① critical path
= instruction + register + mux + ALU + mux + register (write)

	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

immediate arithmetic opi rt rs imm



① critical path
= register + ALU + MUX

OR = sign extend + mux + ALU + MUX

occurs concurrently.

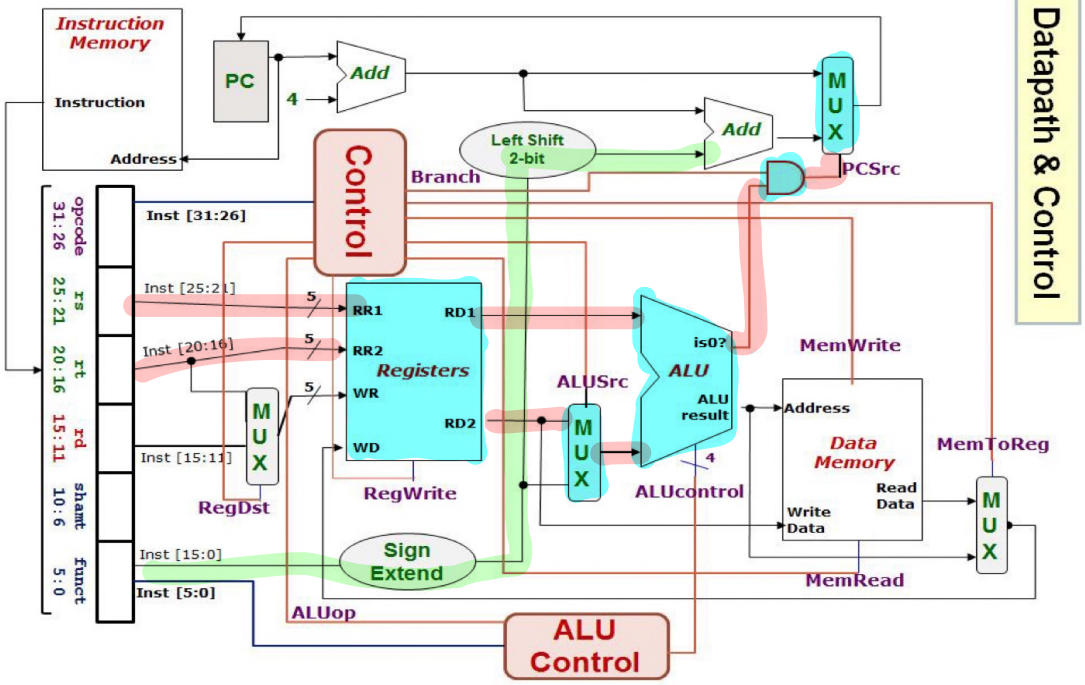
Compare
read register vs sign extension + MUX.

branching beq, bne rs rt imm

① critical path = instruction + register + MUX + ALU + AND + MUX (PCSrc)

or

② sign extend + shift + add + MUX (PCSrc)

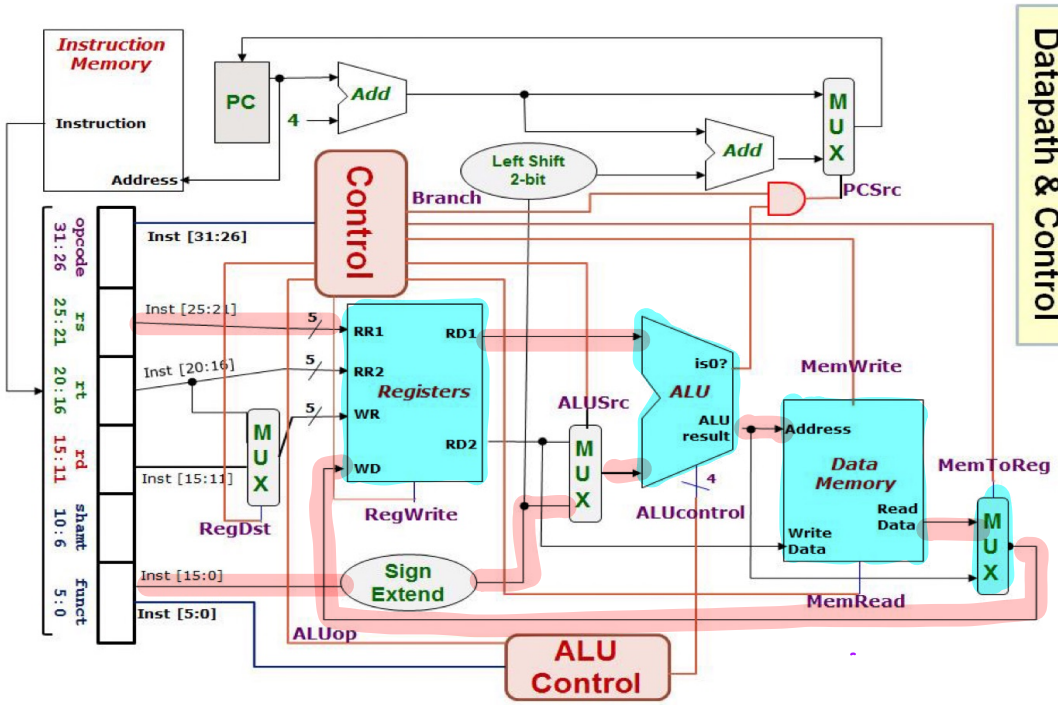


	RegDst	ALUSrc	MemTo Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop	
								op1	op0
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

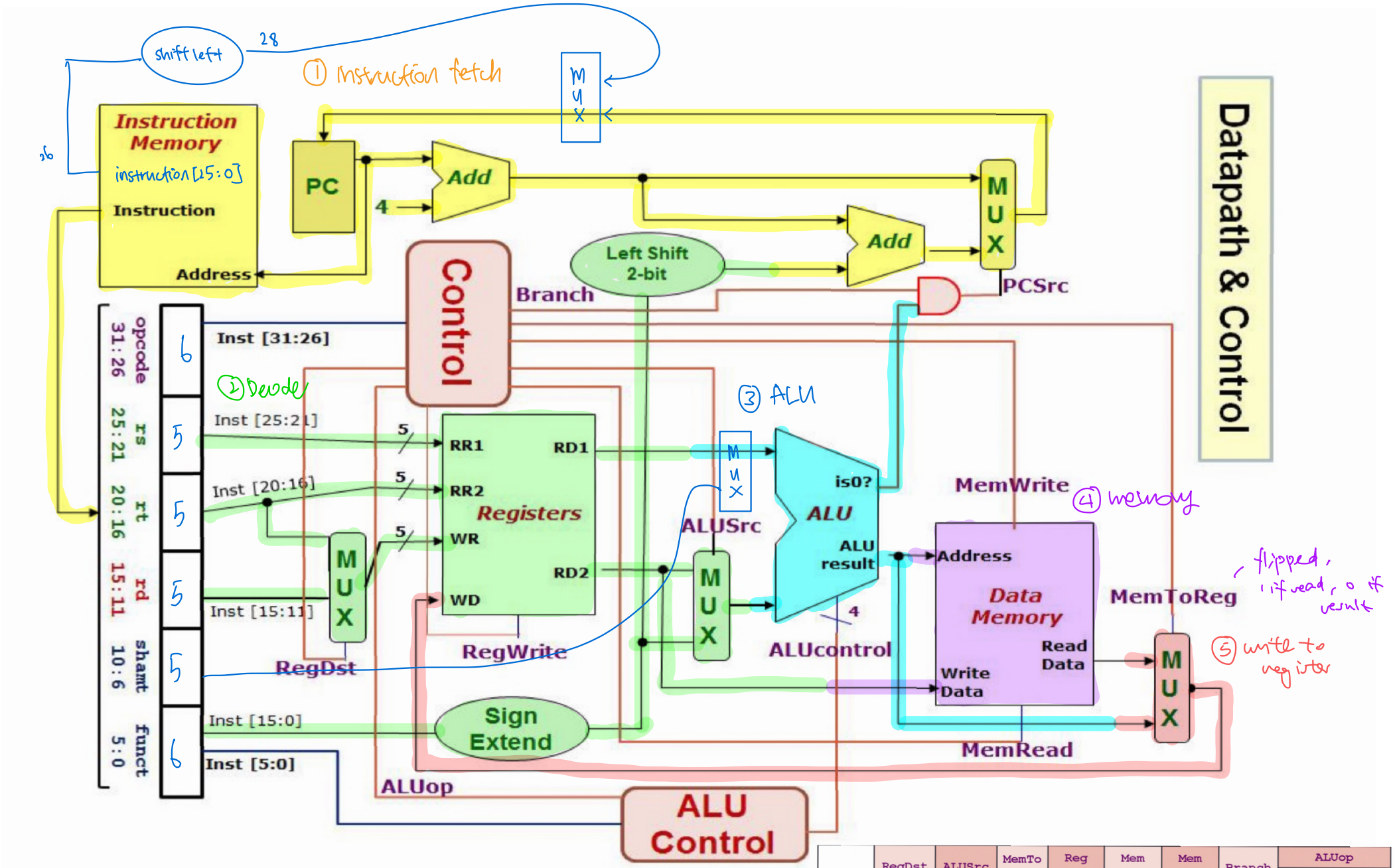
load/store lw/sw rt imm(rs)

paths = instruction + register (or) sign extend + MUX + ALU + memory

LW { + MUX (MemToR) + register

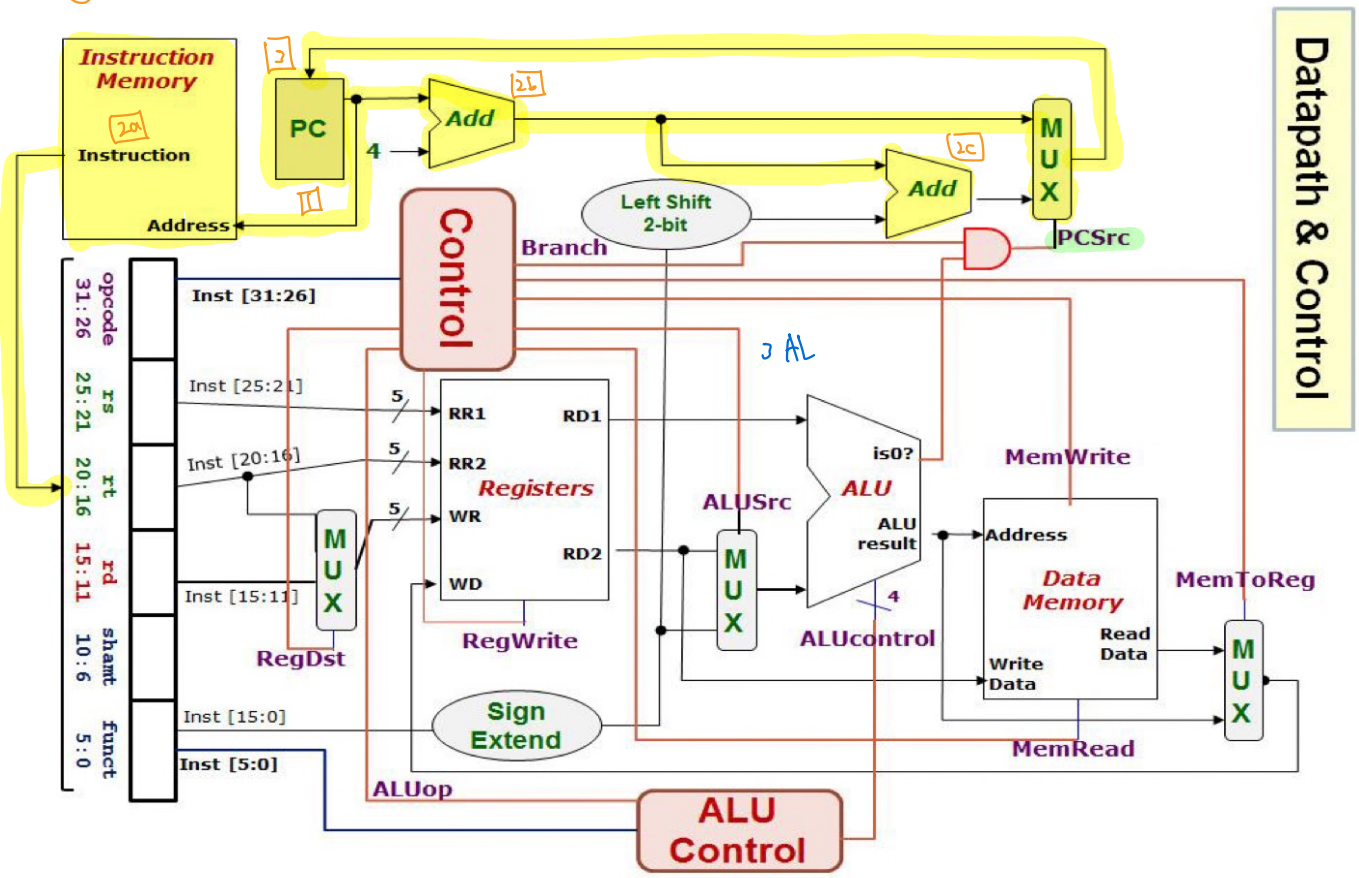


4. Datapath



	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

① instruction fetch stage



① Instruction Fetch

process

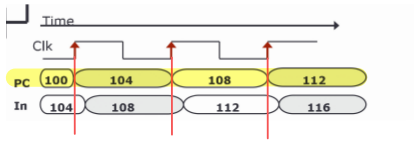
1) During first half of clock period PC is read, sent to memory

2a) instruction is loaded from instruction memory and sent to decode stage

2b) adder adds PC value. PC+4 is also added to sign extended 16 bit immediate.

2c) multiplexer chooses which (PC+4 or PC+4+immediate×4) to return depending on PCSrc

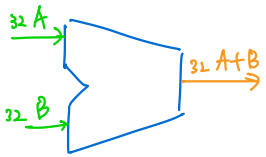
3) PC is updated with new value at the next rising edge



components

1) Adder

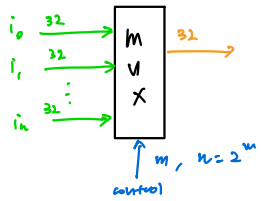
↳ combinational logic circuit to implement addition of two numbers



2) multiplexer

↳ circuit that selects one input from multiple based on m bit control signal

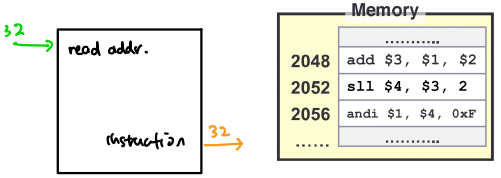
↳ select i^{th} line if control signal = i



3) instruction memory

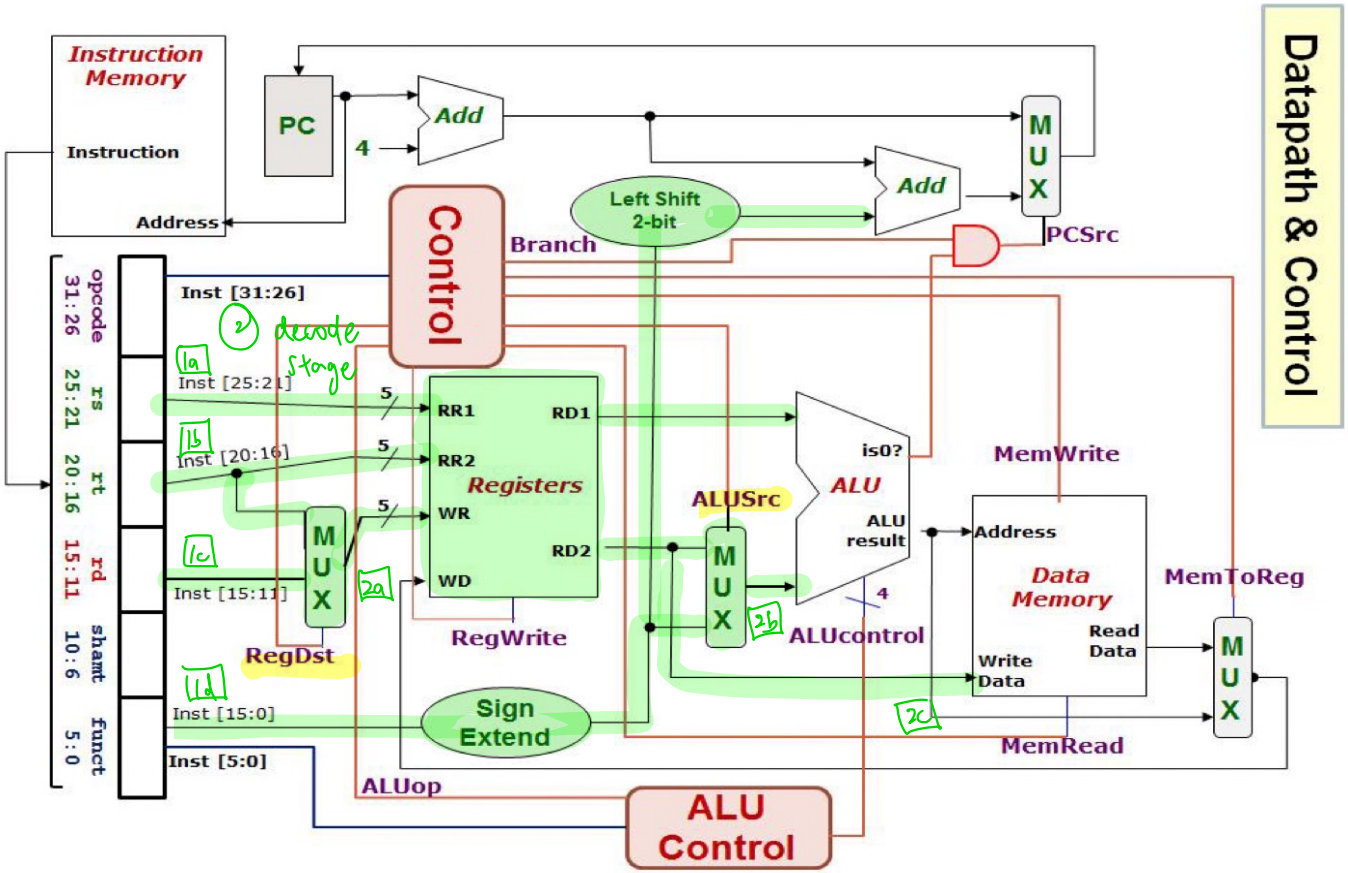
↳ sequential circuit that has internal state to store information

↳ given instruction address as input, outputs content (instruction) at address



4) PC (program counter)

↳ special register that stores address of current instruction being executed



② Decode stage

Process

1a. value in rs is loaded to RR1 to RD1 and output

1b. value in rt is loaded to RR2, MUX and RD2 and output

1c. value in rd is loaded to MUX1

1d. 16 bit immediate is sign extended and loaded to MUX2, as well as left shifted 2 bits and loaded to adder at fetch stage

2a. MUX1 decides between rd and rt to load into WR using RegDst signal. 0 ⇒ rt, 1 ⇒ rd

2b. MUX2 decides between RD2 and immediate value to send as output using ALUSrc signal.

↳ 0 ⇒ RD2, 1 ⇒ immediate

R mode: op | rs | rt | rd | shift

I mode: op | rs | rt | immed

op: read

2c. RD2 signal is passed to Data memory, for writing to memory if needed.

Components

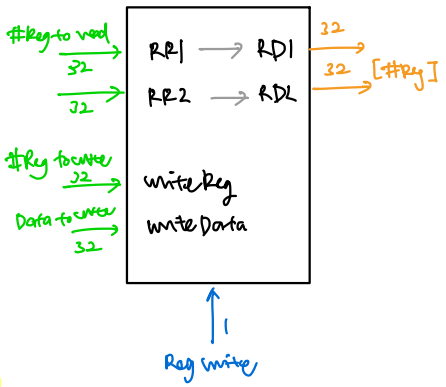
1) register file

↳ a collection of 32 registers

↳ each register is 32 bits wide, can be read/written by specifying reg. number

↳ read at most two registers per instruction, write at most one register per instruction

↳ RegWrite is a control signal to indicate writing of register : (True, 0 False)

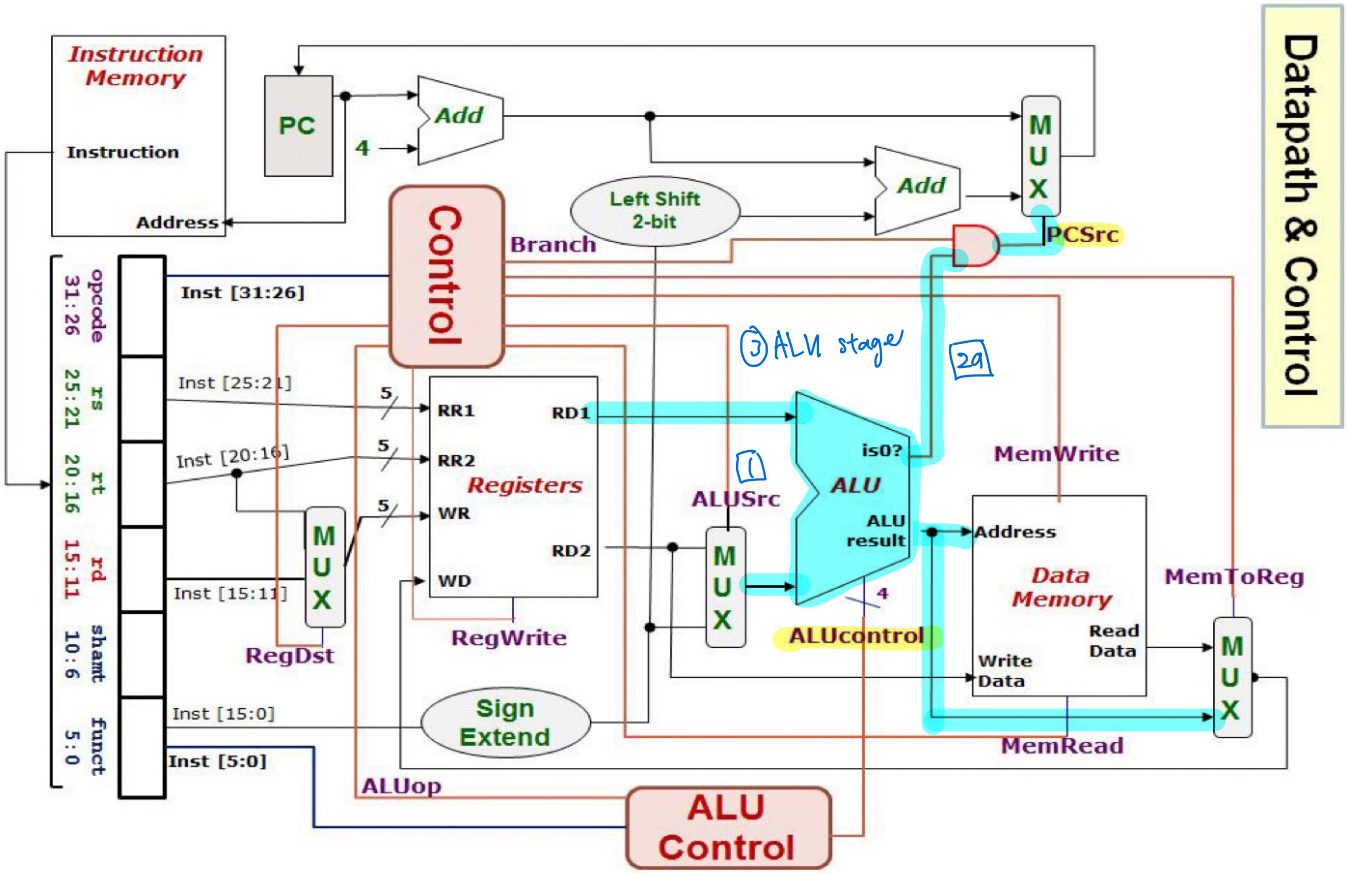


2) sign extend

↳ circuit to extend 16 bit signal to 32 bits

3) 2-bit left shift

↳ circuit to shift 32 bit signal



(3) ALU computation

process

1. operands are read into ALU

2a. ALU does op1 - op2 to test if op1 == op2. signal is sent to AND gate to form control signal PCSrc ~~is~~ control output, to decide if next instruction is PC+4 or PC+4 + immediate x4

2b. ALU does op1 + op2, sends it to memory as well as MUX for later decision of whether to send read data from memory or result directly from ALU.

components

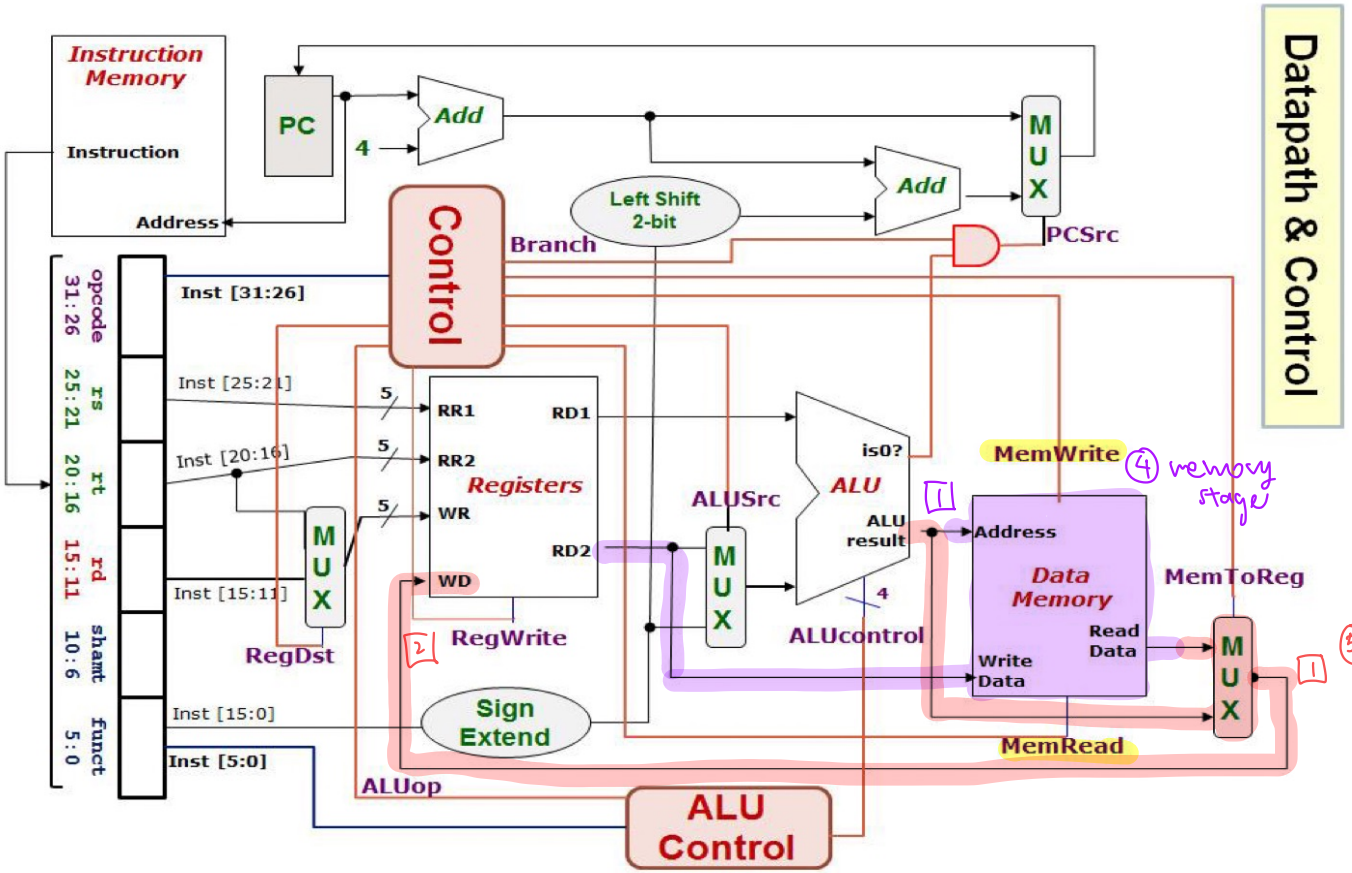
1) Arithmetic logic unit

↳ combinational circuit to implement arithmetic and logic operations

↳ uses 4 bit ALUctl control signal to indicate operation

↳ outputs 32 bit result and 1 bit op1 == op2 result

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



4 memory stage

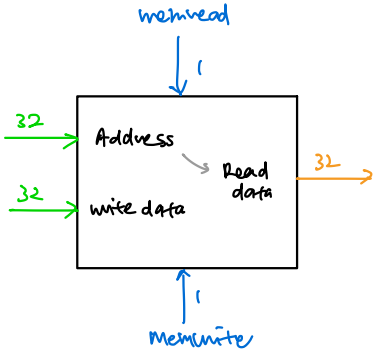
1. output from ALU is read as an address and output to RD to MUX.
2. if memWrite = 1, then output from RD2 is written to memory at address from ALU.

5 write stage

1. MUX routes correct result to register file depending on MemToReg signal
2. written in register WR determined at decode stage

components

1) Data memory
   - storage element for data
   - takes in memory address from ALU and data to be written from RD2
   - depending on memWrite / memRead signals, will write to memory or output to read data. only 1 can be asserted at a time. so signal is default 1 0000 ...)



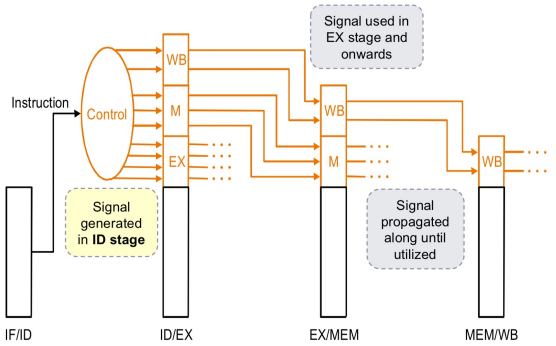
13 - Pipelining

① stages and datapath (naive implementation)

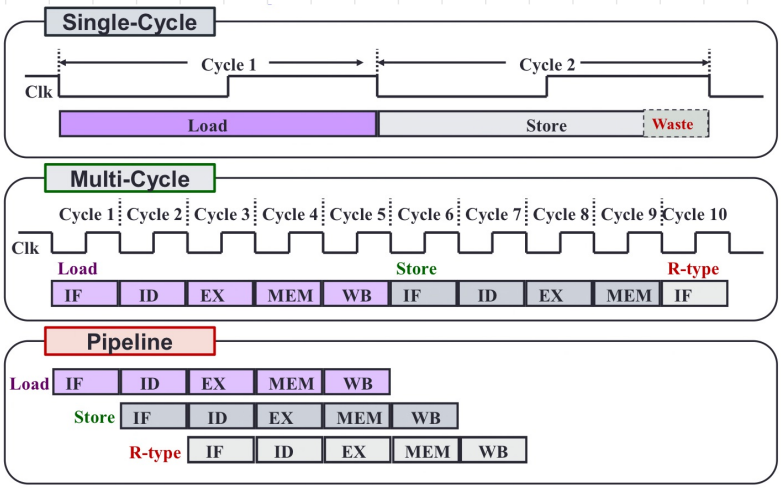
1) intuition: to keep instructions separate, we use special 'pipeline registers' to store and pass forward relevant information to the next execution stage

2) control: pass on relevant signals through pipeline registers

	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



③ Performance of processors



single cycle

$CT = \sum T_i$ $\quad T = n_{instructions} \cdot CT$

$cycles = n_{instructions}$

multicycle

$CT = \max(T_i)$ $\quad T = n_{instructions} \cdot \text{average CPI} \cdot CT$

cycles = variable ⇒ average

pipelined

$CT = \max(T_k) + T_{overhead}$ $\quad T = cycles \cdot CT$

$cycles = n_{instructions} + \underbrace{N-1}_{\text{stages wasted to fill up pipeline}}$

$$Speedup_{pipeline} = \frac{Time_{seq}}{Time_{pipeline}}$$

$$= \frac{I \times \sum_{k=1}^{N} T_k}{(I+N-1) \times (\max(T_k) + T_d)}$$

$$= \frac{I \times N \times T_k}{(I+N-1) \times T_k}$$

$$\approx \frac{I \times N \times T_k}{I \times T_k}$$

$$\approx N$$

Conclusion: Pipeline processor can gain **N** times speedup, where **N** is the number of pipeline stages

delayed branch

Observation:

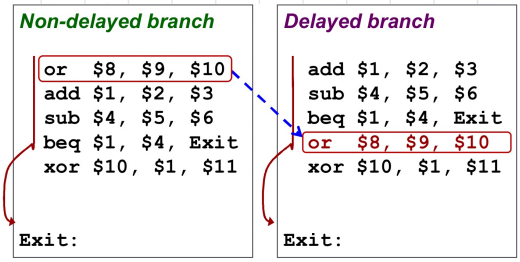
- Branch outcome takes **X** number of cycles to be known
  - ➔ **X** cycles stall ⇒ no. of 'slots' to put after branch

Idea:

- Move **non-control dependent instructions** into the X slots following a branch
  - Known as the **branch-delay slot**
- ➔ These instructions are executed **regardless of the branch outcome**

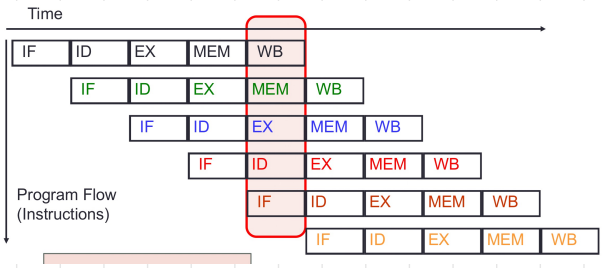
↳ either better off or no difference

eg.



- The "or" instruction is moved into the delayed slot:
  - Get executed regardless of the branch outcome
  - ➔ Same behavior as the original code!

④ Hazards



IF → ID → EX → MEM → WB

Instruction mem | Reg. File | ALU | memory | Reg. file

Reg. File: read values

ALU: branch decision → branch (IF)

memory: jump / branch decision → branch (IF)

Reg. file: write

1. Spot RAW dependencies (affects next two instructions)

write, read, read

no forwarding — write in first half, read in second

IF ID EX M WB
X X IF ID EX M WB
stall 2

normal forwarding — ALU out ⇒ ALU in for next cycle

IF ID EX M WB
IF ID EX M WB
no stall

any $rd (write)
any $rs (read)

LW forwarding — MEM out ⇒ ALU in for next cycle

LW IF ID EX M WB
next X IF ID EX M WB
stall 1

LW $rt (write)
any $rs (read)

2. Spot jumps and branches. Based on conditions and RAW dependencies, stall accordingly

③ branching — Decision in stage 3, next PC addr in stage 4, so branch in stage 5. Addr supplied by pipeline register.

beq IF ID EX M WB
next X X X IF ID EX M WB
stall 3

RAW on pre-beq same as non-branching.

early branching w/ forwarding

normal — Decision & next PC addr made in stage 2 so IF in stage 3, supplied by pipeline reg.

①

beq IF ID EX M WB
next X IF ID EX M WB
stall 1

early branching w/o forwarding

① normal — Decision & next PC addr made in stage 2 so IF in stage 3, supplied by pipeline reg.

beq IF ID EX M WB
next X IF ID EX M WB
stall 1

③ RAW / LW RAW — result written in stage 5, read in second half. next PC calculated in stage 2. Branch in 3.

write $rt IF ID EX M WB
beq $rt X X IF ID EX M WB
next stall 3 X IF ID EX M WB

③ LW RAW — Dependent value retrieved in stage 4. Then forward from pipeline register to ID of next cycle. Stall 2. Branch decision and next PC calculated in stage 2. stall 1. Branch in stage 3.

LW $rt IF ID EX M WB
beq $rt X X IF ID EX M WB
next stall 3 X IF ID EX M WB

② RAW — Dependent value calculated in stage 3. Forward from pipeline reg. to next ID. Stall 1. Then stall 1 for ID stage to make branch decision & calculate new PC. Branch in stage 3.

write $rt IF ID EX M WB
beq $rt X IF ID EX M WB
next stall 2 X IF ID EX M WB

Branch prediction

- Simple prediction:
  - All branches are assumed to be **not taken**
  - → Fetch the successor instruction and start pumping it through the pipeline stages
- When the actual branch outcome is known:
  - **Not taken**: Guessed correctly → No pipeline stall
  - **Taken**: Guessed wrongly → Wrong instructions in the pipeline → **Flush** successor instruction from the pipeline

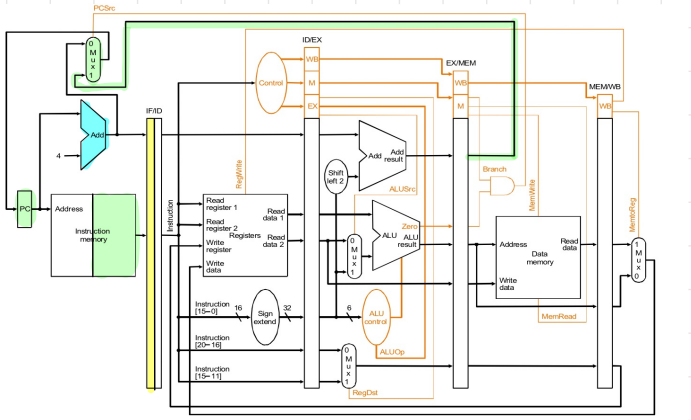
if early branching → with forwarding → flush → 1 / 2 RAW w/ R / 3 RAW w/ lw

→ w/o forwarding → flush → 1 / 3 RAW

Normal: flush 3.

Naive implementation

1. Instruction fetch



First half

MUX chooses next instruction, passes to IM.

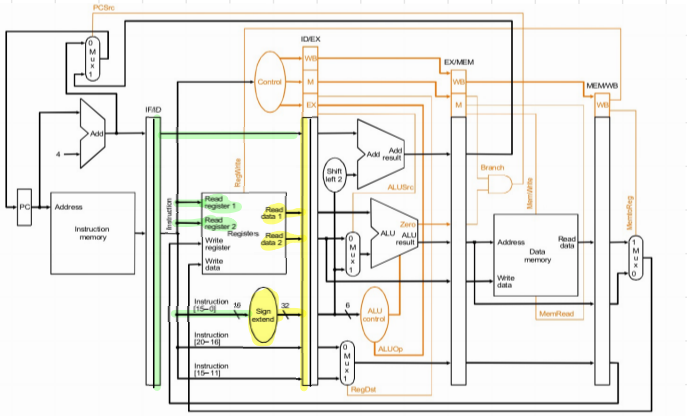
Adder calculates PC+4

Second half

IF/ID register recieves : instruction read

PC+4

2. Instruction decode



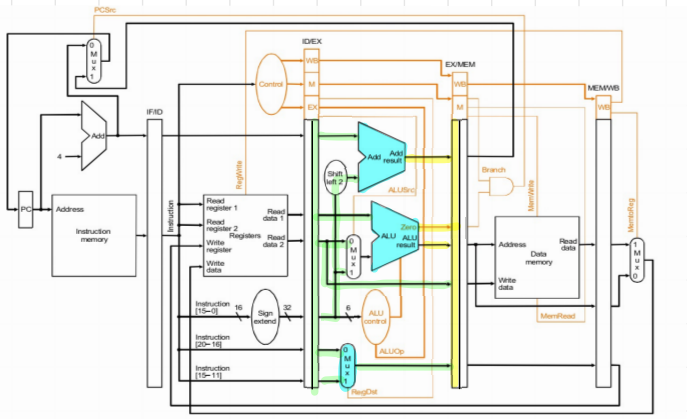
First half

IF/ID supplies register numbers / 16 bit immediate

Second half

Data values read from register file
sign extended immediate
PC+4 pass forward
WR pass forward
} ID/EX recieves

3. Execution



First half

ID/EX supplies : Data values read from register file → ALU

sign extended immediate
PC+4 pass forward
WR pass forward
} PC+4+imm computed

WR or Rt (for SW) chosen by MUX

Second half

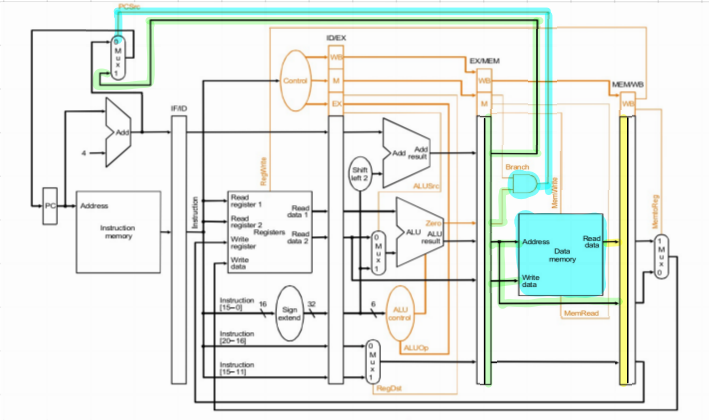
EX/MEM recieves : PC+4 + 4 x imm

ALU result

is Zero? signal

Data read 2 from RF

4. mem stage



First half

EX/MEM supplies : PC + 4 + 4 x imm for next inst.

ALU result for addr.

RD2 writing into mem

ALU result is zero?

memory reads or writes data — rather than in stage 3, to reduce instr. cycle time, since constrain bottleneck of one stage's does too much

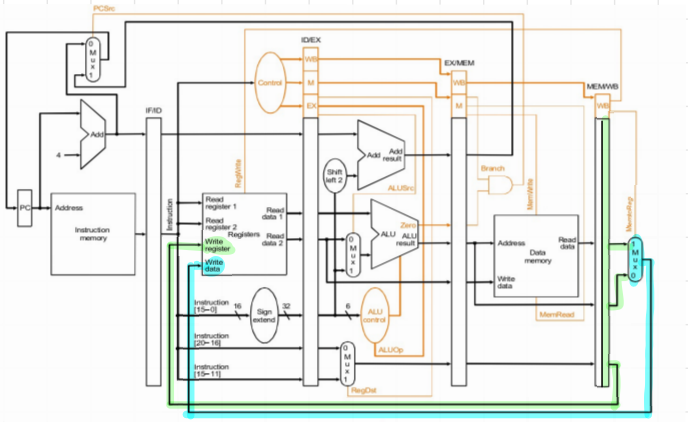
is zero signal used by AND to decide branch

Second half

MEM/WB receives : data from memory

ALU result

5. WB stage



First half

MEM/WB Supplies : ALU result & data from mem

WR register

MUX chooses and writes to RF

Second half

nothing (RF used by stage 2 to read)

14- Caching

① hits & misses

Hit: Data is in cache (e.g., **X**)
- Hit rate: Fraction of memory accesses that hit
- Hit time: Time to access cache

Miss: Data is not in cache (e.g., **Y**)
- Miss rate = 1 – Hit rate
- Miss penalty: Time to replace cache block + hit time

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

- **Compulsory misses** (ALL)
  - On the first access to a block; the block must be brought into the cache @ first call
  - Also called cold start misses or first reference misses
- **Conflict misses** (SA) (direct)
  - Occur in the case of direct mapped cache or set associative cache, when several blocks are mapped to the same block/set i.e. something alr there
  - Also called collision misses or interference misses
- **Capacity misses** (ALL)
  - Occur when blocks are discarded from cache as cache cannot contain all blocks needed
    i.e. was once there, just removed due to other call ⇒ cache size dependent
    size ↑, miss ↓

---

② write policy (sw instructions)

write request → miss
↓
hit → write to cache

Cache and main memory are inconsistent
- Modified data only in cache, not in memory!

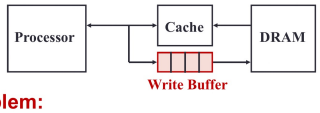
Solution 1: Write-through cache
- Write data both to cache and to main memory

Solution 2: Write-back cache
- Only write to cache
- Write to main memory only when cache block is replaced (evicted)

Write Miss option 1: **Write allocate**
- Load the complete block into cache
- Change only the required word in cache
- Write to main memory depends on write policy

Write Miss option 2: **Write around**
- Do not load the block to cache
- Write directly to **main memory only**

write through cache (both)



Problem:
- Write will operate at the speed of main memory!

Solution:
- Put a write buffer between cache and main memory
  - Processor: writes data to cache + write buffer
  - Memory controller: write contents of the buffer to memory (kinda like async)

write back cache (cache, memory @ evictions)

Problem:
- Quite wasteful if we write back every evicted cache blocks

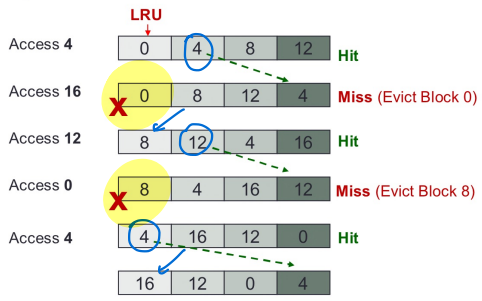
Solution:
- Add an additional bit (**Dirty bit**) to each cache block
- Write operation will change dirty bit to 1
  - Only cache block is updated, no write to memory
- When a cache block is replaced:
  - Only write back to memory if dirty bit is 1 ⇒ bit reset

↳ so writing from cache to memory only when necessary

---

③ Block replacement policy (SA or FA caches)

Least Recently Used policy in action:
- 4-way SA cache
- Memory accesses: **0 4 8 12 4 16 12 0 4**



1. write down order of memory accesses
2. write out cache access horizontally

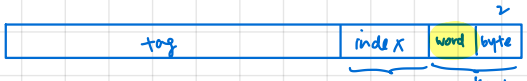
words 0-3

index 0 | I 0-3 | I 8-11 | ...
index 1 | I 4-7 | I 12-16 | ...

④ Mapping policy & Cache types

Direct mapped cache

how it works one row, one set.



$$\text{no. of cache blocks} = \frac{\text{total bytes}}{2^N} = 2^m$$

cache block size $= 2^N$ bytes

1. calculate encoding: tag, index, block, word, byte.
2. create table

	word 0	word 1	word 2	word 3
index 0				
index 1				
⋮				

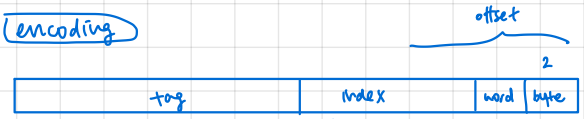
3. look at word/byte bit to see where it will go
4. Determine hit / miss & collisions

Set Associative Cache

how it works

n-way: cache made up of distinct rows of sets, each set able to hold n blocks
⇒ reduce conflict misses

encoding



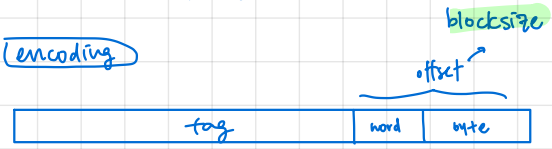
no. of sets = size // (no. of blocks × block size)

Fully associative cache

how it works

↳ no mapping policy. Just added.

encoding



7. Boolean algebra

① Boolean values and operators

1) operator precedence NOT (') > AND (·) > OR (+)

2) laws & theorems

Identity laws	Dual form
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	Dual form
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)

3) duality: if the AND/OR operators & 1/0 in a boolean equation are swapped, the equation remains valid.
↳ logically equivalent. ⇒ useful in proofs or simplification

② Standard forms

1) building blocks

literal: single boolean variable
x, x'

product term: logical product of literals or a single literal
x·y·z', x

sum term: logical sum of literals or a single literal
x, x + y' + z

2) combinations

sum of products: a single product term or a logical sum of several
x, x + y, x + y·z, y·z + x·y'

product of sums: a single sum term or logical product of several sum terms
x, x·(y + z), (x + y)·(x + z)

③ Canonical standard forms

1) minterms & maxterms of n variables → contain n literals from all the variables

product term ↔ sum term (complements)

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

notice its inversion when 0,0

complements

2) canonical SOP and POS
↳ SOP of miniterms / POS of max terms

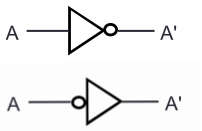
3) conversion between SOP and POS ⇒ complement

$\Sigma(1,4,5,6,7) = \Pi M(0,2,3)$

8. Logic circuits

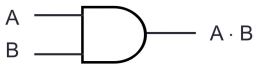
① Logic gates and truth tables

NOT



A	A'
0	1
1	0

AND

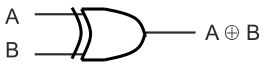


A	B	A · B
0	0	0
0	1	0
1	0	0
1	1	1

OR

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

XOR

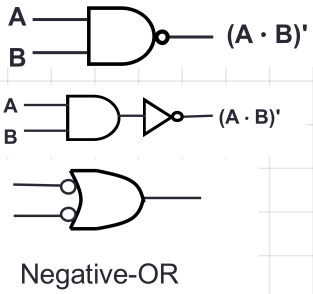


A	B	A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

XNOR

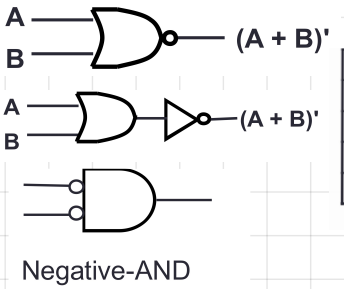
A	B	(A ⊕ B)'
0	0	1
0	1	0
1	0	0
1	1	1

NAND



A	B	(A · B)'
0	0	1
0	1	1
1	0	1
1	1	0

NOR

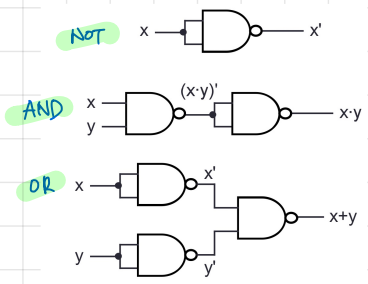


A	B	(A + B)'
0	0	1
0	1	0
1	0	0
1	1	0

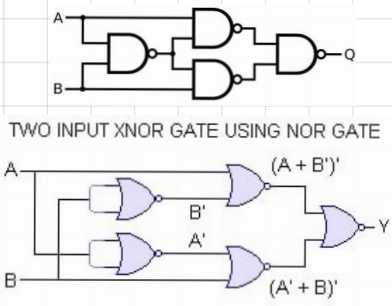
② Complete logic sets and universal gates

1) complete sets of logic (AND / OR / NOT) with NAND & NOR, also (AND , NOT)

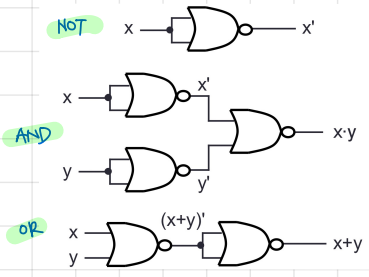
NAND



XOR

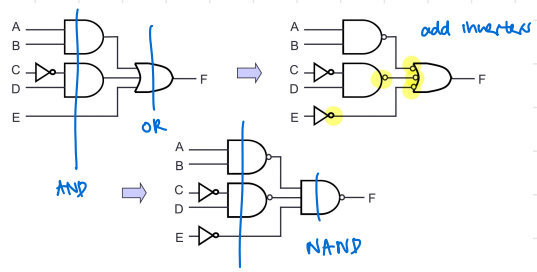


NOR

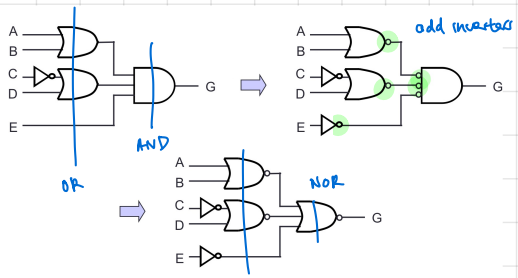


2) 2-level circuits (NOT discounted)

SOP AND/OR | NAND



POS OR/AND | NOR

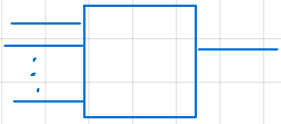


10. combinational circuit

① circuit delays

↳ logic gate w/ delay t ; inputs stable at $t_1, t_2 \ldots$

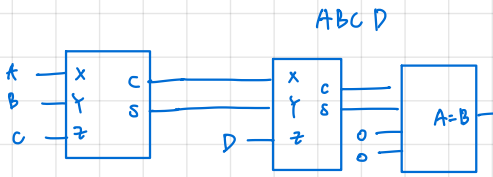
⇒ time at which output is stable is $\max(t_1, t_2 \ldots) + t$



$\max(t_1 \ldots t_i) + t$

② problem solving: common cases

1. to count the number of 1s: add all positions eg. using full adders + magnitude comp.



checks that no 1's

2. To multiply / divide ⇒ do right or left shifts

3. when examining truth table, pay attention to adjacent output = 1 terms to identify what matters & what doesn't

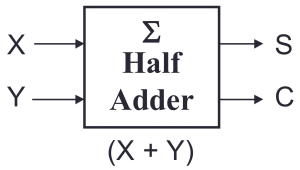
4. systematically look at subgroups □ □ □ □, commonly first / last n

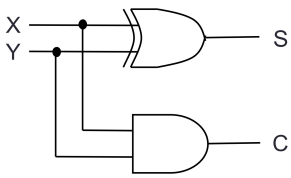
5. when a lot eg $\Sigma m(\ ,\ ,\ ,6,7\ldots) =$ consider using SOP for F' instead, then use ▷o or NOR gate

000100
000110

---

half adder implements X + Y → sum + carry



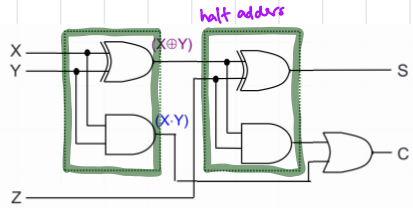


X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$C = X \cdot Y$

$S = X \oplus Y$

full adder implements X + Y + Z (carry) → sum +



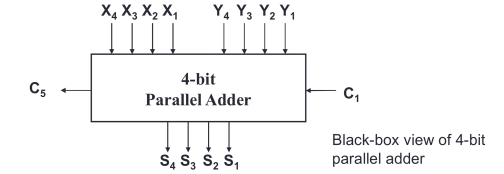
$C = X \cdot Y + (X \oplus Y) \cdot Z$

$S = X \oplus Y \oplus Z$

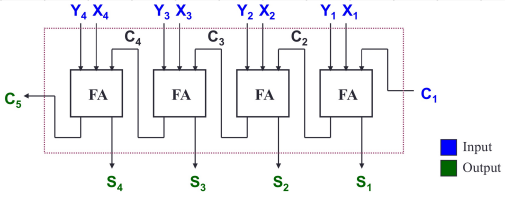
X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

---

parallel adders 9 bit in, 5 bits out. normally implemented by cascading



Black-box view of 4-bit parallel adder



BCD to excess 3 converter converts 0 to 9 to excess 3 representation (BCD + 3)

	BCD				Excess-3			
	A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0
10	1	0	1	0	X	X	X	X
11	1	0	1	1	X	X	X	X
12	1	1	0	0	X	X	X	X
13	1	1	0	1	X	X	X	X
14	1	1	1	0	X	X	X	X
15	1	1	1	1	X	X	X	X

$W = A + B \cdot C + B \cdot D$

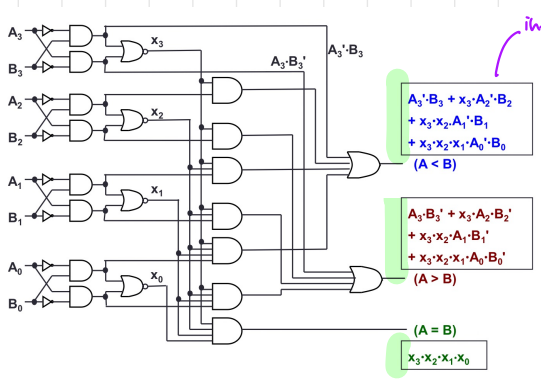
$X = B' \cdot C + B' \cdot D + B \cdot C' \cdot D'$

$Y = C \cdot D + C' \cdot D'$

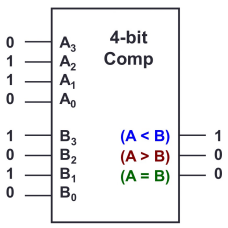
$Z = D'$

magnitude comparators compares unsigned values to check if $A > B$, $A = B$, $A < B$

↳ to consider usage, see if one column always one (i.e. >), or equal, etc.



intuition: bit-by-bit comparison

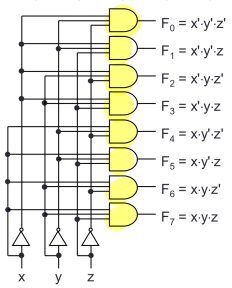


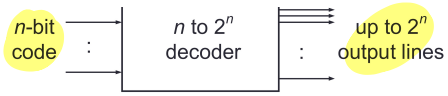
② Decoders

II. MSI circuits

how it works : selects an output line from 2^n of them using a n-bit input, $A \cdot B' \cdot C' \Rightarrow$ AND (only 1 on, rest off)

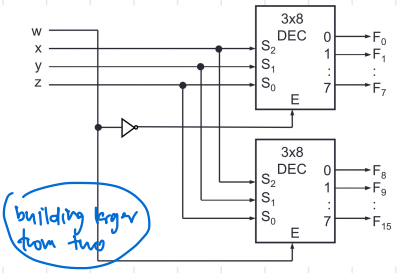
x	y	z	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

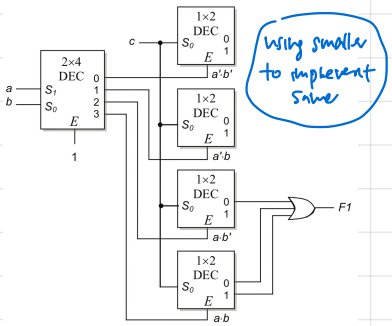




"composite" decoders : use of another bit & gates to choose between decoders and their output ones using enable

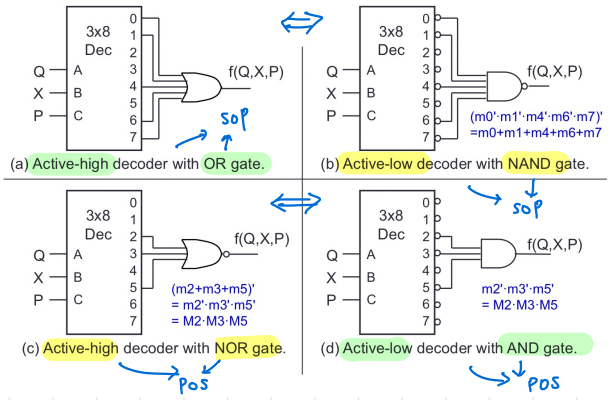
↳ can even use decoder to choose decoder





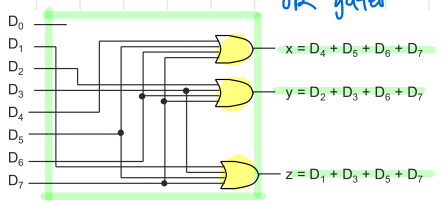
implementing functions boolean function in SOP, use gates

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



③ Encoders

how it works compresses 2^n input to n bits using OR gates



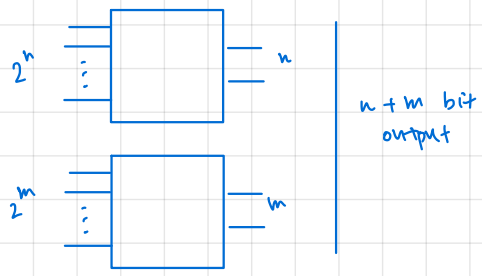
An 8-to-3 encoder

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

priority encoders if two or more inputs = 1, input w the highest priority takes 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

constructing larger encoders



④ Multiplexers

how it works 2^n input lines, n selection lines, m (identical) output lines. "$2^n : m$ mux."

↳ implemented by AND of input & decoder selection, then OR.

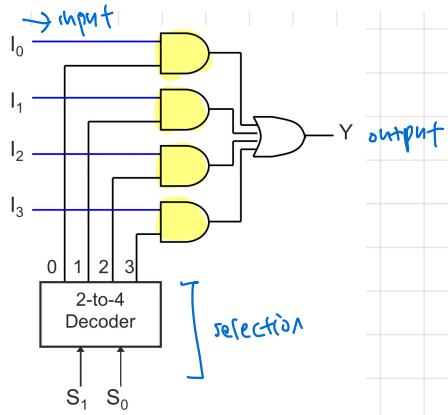
Output of multiplexer is

"sum of the (product of *data lines* and *selection lines*)"

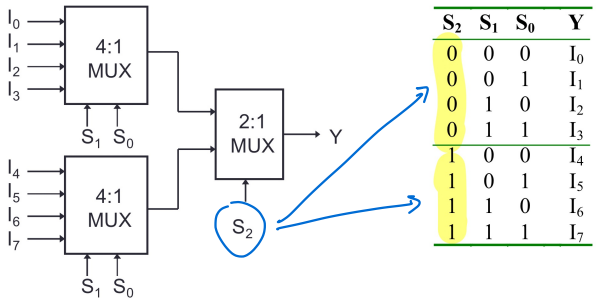
Example: Output of a 4-to-1 multiplexer is:

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

0	1
1	0
1	1



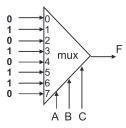
constructing larger multiplexers breaking up bits to choose ⇒ use MSB



S_2	S_1	S_0	Y
0	0	0	I_0
0	0	1	I_1
0	1	0	I_2
0	1	1	I_3
1	0	0	I_4
1	0	1	I_5
1	1	0	I_6
1	1	1	I_7

implementing functions boolean SOP functions

1. Express in sum-of-minterms form.
   Example:
   $F(A,B,C) = A'\cdot B'\cdot C + A'\cdot B\cdot C + A\cdot B'\cdot C + A\cdot B\cdot C'$
   $= \Sigma\, m(1,3,5,6)$
2. Connect n variables to the n selection lines.
3. Put a '1' on a data line if it is a minterm of the function, or '0' otherwise.



This method works because:

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

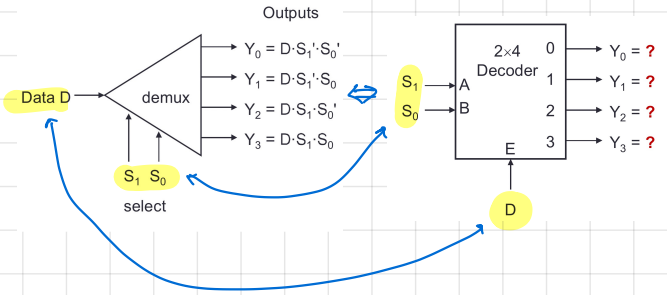
Supplying '1' to I_1, I_3, I_5, I_6, and '0' to the rest:

Output = $m_1 + m_3 + m_5 + m_6$

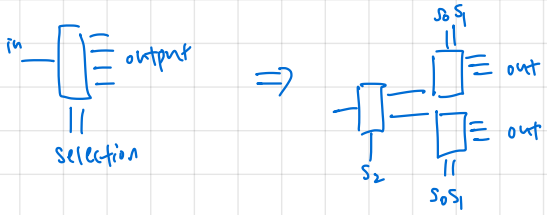
⑤ Demultiplexers

how it works given data input and n bit selection line, directs data (1 bit) to one of 2^n lines.

↳ identical to encoder, where data ≈ enable



constructing comp. demultiplexers

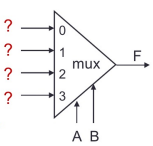


using smaller multiplexers

3. Draw the truth table for function, by grouping inputs by selection line values, then determine multiplexer inputs by comparing input line (C) and function (F) for corresponding selection line values.

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	

regardless of C, F = 1
F = 0
regardless of C, F = 0
F = C'



12. sequential circuits

① analysis of sequential circuits

↳ sequential circuits : built from logic gates and flip flops
↳ intuition behind sequential circuit: for outputs that are previous-state dependent eg. vending machine

1. identify and derive boolean expressions for flipflop inputs and system outputs
2. derive state equations for flip flop states
3. derive state table ⇒ m flip flops (states) and n inputs ⇒ 2^{m+n} rows
   ↳ fill in inputs & states combinations
   ↳ use flip flop input functions to get inputs eg. KA, JA ...
   ↳ use characteristic tables & output function to get new states & outputs
4. draw state diagrams

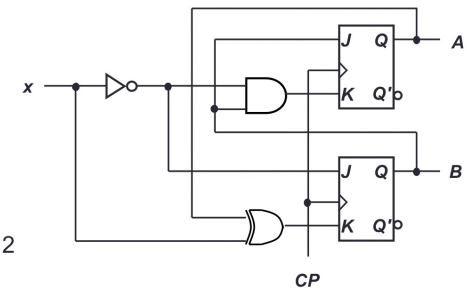


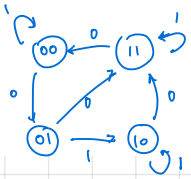
Figure 2

JA = B **JB = x'**
KA = B·x' **KB = A'·x + A·x' = A ⊕ x**

Fill the state table using the above functions, knowing the characteristics of the flip-flops used.

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

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



Obtain the flip-flop input functions from the circuit:

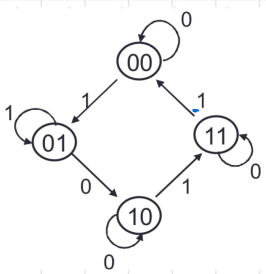
JA = B **JB = x'**
KA = B·x' **KB = A'·x + A·x' = A ⊕ x**

↳ sinks & self connection
state that once a circuit enters, never moves out

↳ if circuit is able to (without inputs) move from an invalid to valid state in a finite no. of steps

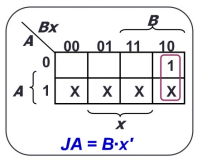
② Design of sequential circuits

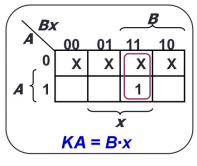
1. From state diagram identify no. of states (flipflops), inputs and outputs
2. Fill in state table
   ↳ fill in combinations of states & inputs
   ↳ from state diagram fill in next states
   ↳ using excitation table, fill in flip flop inputs
   ↳ use k maps to derive flip flop input expressions, implement

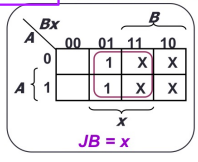


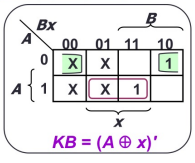
From state table, get flip-flop input functions.

Present state		Input	Next state		Flip-flop inputs			
A	B	x	A^+	B^+	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



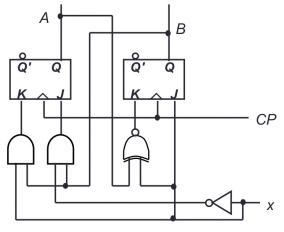






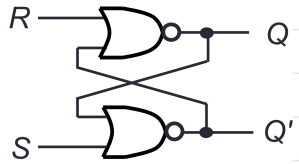
Flip-flop input functions:
JA = B·x' **JB = x**
KA = B·x **KB = (A ⊕ x)'**

Logic diagram:



S-R

ungated SR latch

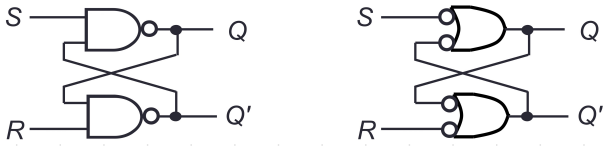


S	R	Q(t+1)	Comments
0	0	Q(t)	No change
0	1	0	Reset
1	0	1	Set
1	1	?	Unpredictable

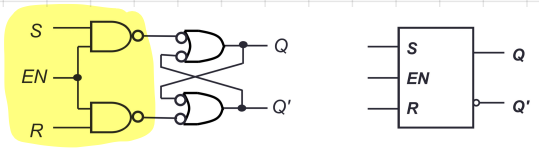
Q	Q^+	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

$Q^+ = ?$

ungated active-low SR latch (inputs are flipped)



gated SR latch / flip flop



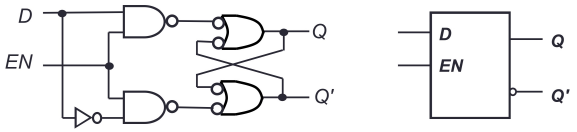
D Direct

gated D latch / flip flop

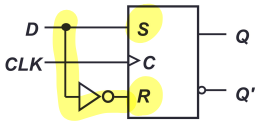
$Q^+ = D$

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

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



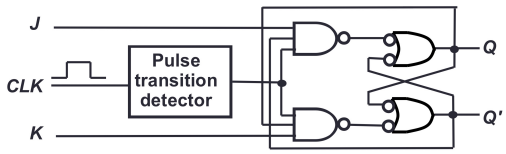
Convert *S-R* flip-flop into a *D* flip-flop: add an inverter.



↳ gets rid of undesirable invalid SR state, but loses ability to 'no change'

JK flip flop

valid version of SR



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

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

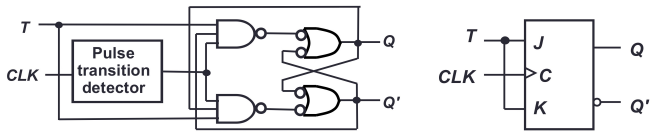
$Q^+ = J \cdot Q' + K' \cdot Q$

↳ has no invalid output. Same as SR, but invalid replaced with toggle.

↳ Q and Q^+ are fed back to pulse-steered NAND gates

T flip flop

Toggle



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

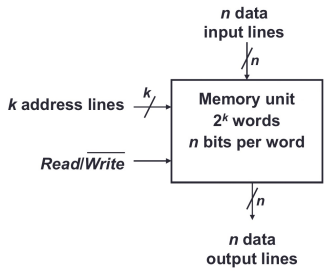
T	Q(t+1)	
0	Q(t)	No change
1	Q(t)'	Toggle

↳ toggles. Formed from JK by tying inputs together.

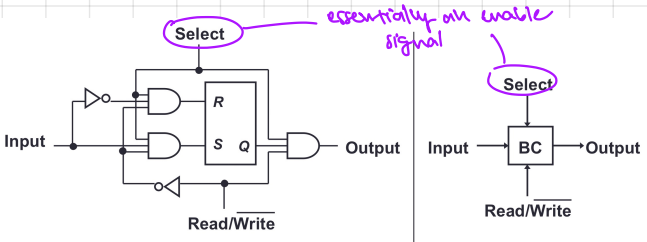
3) memory

1) the memory unit

Memory Enable	*Read/Write*	Memory Operation
0	X	None
1	0	Write to selected word
1	1	Read from selected word



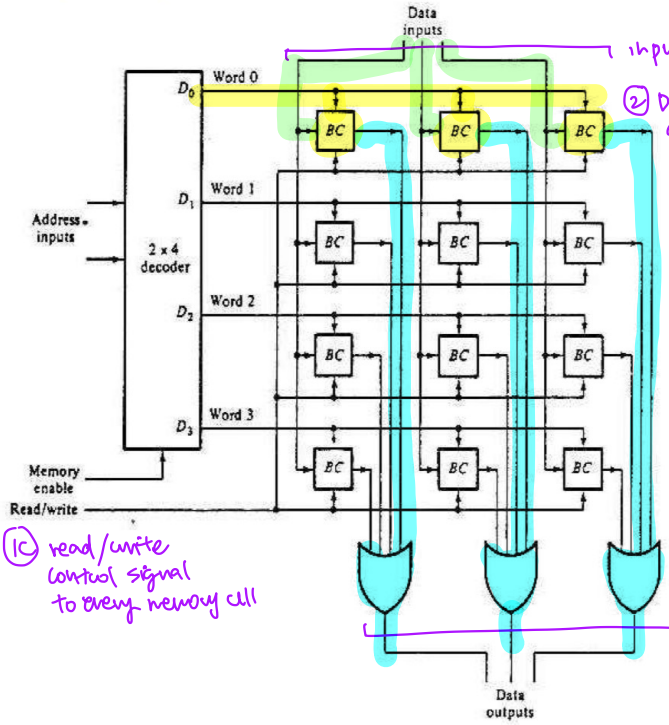
2) memory cells of static RAM : flip flops are used



added wiring around SR flipflop is so that behaviour follows memory control signals

- read = 1 → uncharged
- write = 0 → Q = input
- select → enable

3) arrays of memory cells to build memory



(1A) address into decoder

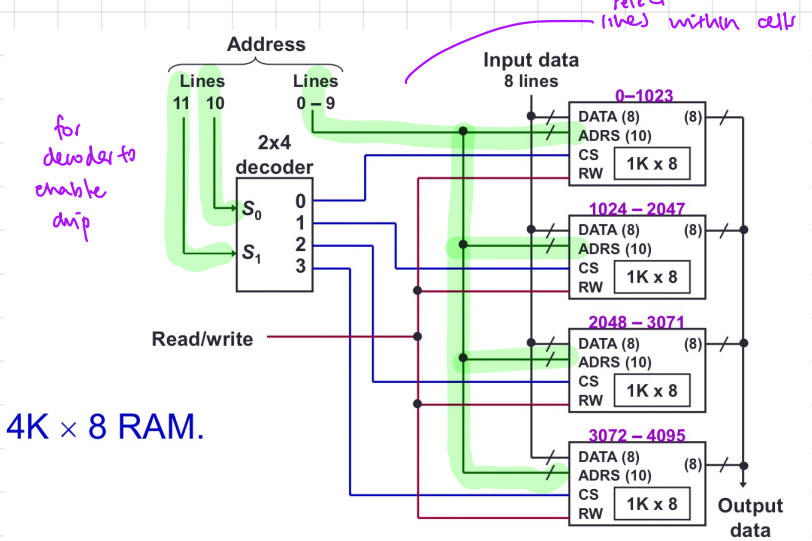
(1B) memory enable to decoder

(1C) read/write control signal to every memory cell

(2) Decoder enables word determined by address by providing select signal

(3) based on Read/write, state changes / uncharged & output to OR gate

4) arrays of memory chips to build larger memory



for decoder to enable chip

4K × 8 RAM.