

PARCIALES



Key Color

2017 C1



Envia tus exámenes a diq.fiu

A-B3h B
realizar las operaciones

$$A+B \quad A-B \quad (-A)+B$$

$A+B$ $A-B$ $(-A)+B$
in case indicated

en cada caso indicar si existe o no una condición fuera de rango (justificar)

b) Considere que A y B son enteros sin signo y obtenga la suma A+B indicando si existe o no una condición de fuera de rango (justificar).

$$a) A = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \end{pmatrix} = \text{Ca2 (77)}$$

$$B = 0 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 = \text{ca2}(-115)$$

$$\begin{array}{r}
 \bullet \quad A + B : \\
 \hline
 & + \\
 & \hline
 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\
 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\
 \hline
 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\
 \downarrow & & & & & & & \\
 & + & & & & & & \\
 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\
 & & & & & & & \\
 \hline
 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0
 \end{array}$$

$$1 \ 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0 = 218 = -38 \Rightarrow A+B = 0010011 = 2 \ 2 (-38)$$

se ve de
nemoj

$$\text{verifico: } A + B = \text{Ca2}(77) + \text{Ca2}(-115) = -77 + 115 = 38 = \text{Ca2}(-38) \quad \checkmark$$

$$\begin{array}{r}
 \bullet A - B : \\
 \hline
 & \begin{array}{r} 0 \\ y \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{array} \\
 & \begin{array}{r} 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{array} \\
 & \hline
 & \begin{array}{r} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{array} \\
 & \downarrow \\
 & \begin{array}{r} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{array} \\
 + & \begin{array}{r} 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{array} \\
 \hline
 & \begin{array}{r} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{array}
 \end{array}$$

$$\begin{array}{r} 1 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\ \hline 1 \ 9 \ 2 \ -6 \ 4 \end{array} \Rightarrow A - B = \text{Ca2}(-64) = 01\ 000\ 000$$

Se va de rompe

$$\text{verifio: } A - B = \text{Ca2}(77) - \text{Ca2}(-115) = -77 - 115 = -192 = 64 = \text{Ca2}(-64) \quad \checkmark$$

Se va de romero

$$\begin{array}{r} \underline{+ A + B :} \quad -1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \rightarrow 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \\ \quad \quad \quad + 0 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \\ \hline \quad \quad \quad 0 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 1 \end{array}$$

$$\begin{array}{r}
 \Rightarrow -10110011 \\
 +01110011 \\
 \hline
 \end{array}
 \equiv +
 \begin{array}{r}
 1111110 \\
 0100110 \\
 01110011 \\
 \hline
 11000000 \\
 \end{array}
 + \begin{array}{r}
 1111111 \\
 0011111 \\
 \hline
 01000000 \\
 \end{array}$$

$$\overline{0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0} = 64 \Rightarrow -A + B = \text{C22 (64)} = 11\ 000000$$

$$\text{verif. } -A + B = -C_{22}(77) + C_{22}(-115) = 77 + 115 = 192 = -64 = C_{22}(64) \quad \checkmark$$

$$b) \begin{array}{r} \begin{array}{ccccccc} 1 & 1 & 1 & & 1 & 1 & \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\ \hline 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \end{array} \\ + \end{array} = \begin{array}{r} + \begin{array}{ccc} 1 & 2 & 9 \\ 1 & 1 & 5 \\ \hline 2 & 9 & 4 \end{array} \\ > 2^8 = 256 \end{array}$$

en $C = 1$ pose même
sin signo es uno
alre he os feero
de razon.

Además, recordar que $A^1_{10} + B^1_{10} = 2941_{10} > 256$, es decir, el resultado de la suma no puede expresarse en un registro de 8 bits, es decir que resulta de varios dos bytes para guardar el resultado de $A + B$. ☺

2) Dados los siguientes números en base 10

$$C = 7.6719 \quad D = 2.5$$

- Representarlos en un sistema 8 bits de punto fijo que reserva 4 bits para la parte fraccionaria
- Representarlos en punto flotante normalizados según la norma IEEE754 para simple precisión
- Obtener la suma $C+D$ operando en punto fijo y luego operando en punto flotante (en este último caso indicar el resultado normalizado)

a) $C = 7,6719$

$$\begin{array}{r} 7 \ 12 \\ \underline{3} \ \underline{2} \\ 8 \end{array} \Rightarrow 7 \frac{1}{10} = 1111_2 \rightarrow \text{ocupa 3 bits} \Rightarrow \text{me quedan 5}$$

$$0,6719 \cdot 2 = 1,3438$$

$$0,3438 \cdot 2 = 0,6876$$

$$0,6876 \cdot 2 = 1,3752$$

$$0,3752 \cdot 2 = 0,7504$$

↳ hasta aquí tengo 4 bits para representar la parte decimal

luego: $C = 7,6719_1 = 01111010_2$

$$D = 2,5$$

$$2 \frac{1}{10} = 001_2$$

$$0,5 \cdot 2 = 1,0$$

$$0,0 \cdot 2 = 0,0$$

$$0,0 \cdot 2 = 0,0$$

$$0,0 \cdot 2 = 0,0$$

luego: $D = 2,5_1 = 00101000_2$

b) $s=1, l=8, m=23$

$$C = 7,6719_1 = 111,1000_2 \xrightarrow{\text{normalizo}} 1,111010 \cdot 2^3 \Rightarrow \begin{cases} l = 12 + 2 = 129 = 10000001 \\ s = 0 \\ m = 111010 \rightarrow 6 bits, me quedan 17 \end{cases}$$

$$1) 0,7504 \cdot 2 = 1,5008 \quad 2) 0,4096 \cdot 2 = 0,8192$$

$$3) 0,5008 \cdot 2 = 1,0016 \quad 4) 0,8192 \cdot 2 = 1,4384$$

$$5) 0,0016 \cdot 2 = 0,0032 \quad 6) 1,4384 \cdot 2 = 1,2768$$

$$7) 0,0032 \cdot 2 = 0,0064 \quad 8) 1,2768 \cdot 2 = 0,5536$$

$$9) 0,0064 \cdot 2 = 0,0128 \quad 10) 0,5536 \cdot 2 = 1,1072$$

$$11) 0,0128 \cdot 2 = 0,0256 \quad 12) 1,1072 \cdot 2 = 0,2144$$

$$13) 0,0256 \cdot 2 = 0,0512 \quad 14) 0,2144 \cdot 2 = 0,4288$$

$$15) 0,0512 \cdot 2 = 0,1024 \quad 16) 0,4288 \cdot 2 = 0,8576$$

$$17) 0,1024 \cdot 2 = 0,2048 \quad 18) 0,8576 \cdot 2 = 1,7152$$

$$19) 0,2048 \cdot 2 = 0,4096 \quad 20) 1,7152 \cdot 2 = 3,4304$$

$$21) 0,4096 \cdot 2 = 0,8192 \quad 22) 3,4304 \cdot 2 = 6,8608$$

$$23) 0,8192 \cdot 2 = 1,6384 \quad 24) 6,8608 \cdot 2 = 13,7216$$

$$25) 1,6384 \cdot 2 = 3,2768 \quad 26) 13,7216 \cdot 2 = 27,4432$$

$$27) 3,2768 \cdot 2 = 6,5536 \quad 28) 27,4432 \cdot 2 = 54,8864$$

$$29) 6,5536 \cdot 2 = 13,1072 \quad 30) 54,8864 \cdot 2 = 109,7728$$

$$31) 13,1072 \cdot 2 = 26,2144 \quad 32) 109,7728 \cdot 2 = 219,5456$$

$$33) 26,2144 \cdot 2 = 52,4288 \quad 34) 219,5456 \cdot 2 = 439,0912$$

$$35) 52,4288 \cdot 2 = 104,8576 \quad 36) 439,0912 \cdot 2 = 878,1824$$

$$37) 104,8576 \cdot 2 = 209,7152 \quad 38) 878,1824 \cdot 2 = 1756,3648$$

$$39) 209,7152 \cdot 2 = 419,4304 \quad 40) 1756,3648 \cdot 2 = 3512,7296$$

$$41) 419,4304 \cdot 2 = 838,8608 \quad 42) 3512,7296 \cdot 2 = 7025,4592$$

$$43) 838,8608 \cdot 2 = 1677,7216 \quad 44) 7025,4592 \cdot 2 = 14050,9184$$

$$45) 1677,7216 \cdot 2 = 3355,4432 \quad 46) 14050,9184 \cdot 2 = 28101,8368$$

$$47) 3355,4432 \cdot 2 = 6710,8864 \quad 48) 28101,8368 \cdot 2 = 56203,6736$$

$$49) 6710,8864 \cdot 2 = 13421,7728 \quad 50) 56203,6736 \cdot 2 = 112407,3472$$

$$51) 13421,7728 \cdot 2 = 26843,5456 \quad 52) 112407,3472 \cdot 2 = 224814,6944$$

$$53) 26843,5456 \cdot 2 = 53687,0912 \quad 54) 224814,6944 \cdot 2 = 449629,3888$$

$$55) 53687,0912 \cdot 2 = 107374,1824 \quad 56) 449629,3888 \cdot 2 = 899258,7776$$

$$57) 107374,1824 \cdot 2 = 214748,3648 \quad 58) 899258,7776 \cdot 2 = 1798517,5552$$

$$59) 214748,3648 \cdot 2 = 429496,7296 \quad 60) 1798517,5552 \cdot 2 = 3597035,1104$$

$$61) 429496,7296 \cdot 2 = 858993,4592 \quad 62) 3597035,1104 \cdot 2 = 7194070,2208$$

$$63) 858993,4592 \cdot 2 = 1717986,9184 \quad 64) 7194070,2208 \cdot 2 = 14388140,4416$$

$$65) 1717986,9184 \cdot 2 = 3435973,8368 \quad 66) 14388140,4416 \cdot 2 = 28776280,8832$$

$$67) 3435973,8368 \cdot 2 = 6871947,6736 \quad 68) 28776280,8832 \cdot 2 = 57552561,7664$$

$$69) 6871947,6736 \cdot 2 = 13743895,3472 \quad 70) 57552561,7664 \cdot 2 = 115105123,5344$$

$$71) 13743895,3472 \cdot 2 = 27487790,6944 \quad 72) 115105123,5344 \cdot 2 = 230210247,0688$$

$$73) 27487790,6944 \cdot 2 = 54975581,3888 \quad 74) 230210247,0688 \cdot 2 = 460420494,1376$$

$$75) 54975581,3888 \cdot 2 = 109951162,7776 \quad 76) 460420494,1376 \cdot 2 = 920840988,3152$$

$$77) 109951162,7776 \cdot 2 = 219902325,5552 \quad 78) 920840988,3152 \cdot 2 = 1841681976,6304$$

$$79) 219902325,5552 \cdot 2 = 439804651,1104 \quad 80) 1841681976,6304 \cdot 2 = 3683363953,2608$$

$$81) 439804651,1104 \cdot 2 = 879609302,2208 \quad 82) 3683363953,2608 \cdot 2 = 7366727906,5216$$

$$83) 879609302,2208 \cdot 2 = 1759218604,4416 \quad 84) 7366727906,5216 \cdot 2 = 14733455813,0432$$

$$85) 1759218604,4416 \cdot 2 = 3518437208,8832 \quad 86) 14733455813,0432 \cdot 2 = 29466911626,0864$$

$$87) 3518437208,8832 \cdot 2 = 7036874417,7664 \quad 88) 29466911626,0864 \cdot 2 = 59033823252,1728$$

$$89) 7036874417,7664 \cdot 2 = 14073748835,5344 \quad 90) 59033823252,1728 \cdot 2 = 118067646504,3488$$

$$91) 14073748835,5344 \cdot 2 = 28147497671,0688 \quad 92) 118067646504,3488 \cdot 2 = 236135293008,6976$$

$$93) 28147497671,0688 \cdot 2 = 56294995342,1376 \quad 94) 236135293008,6976 \cdot 2 = 472270586017,3952$$

$$95) 56294995342,1376 \cdot 2 = 112589990684,2752 \quad 96) 472270586017,3952 \cdot 2 = 944541172034,7904$$

$$97) 112589990684,2752 \cdot 2 = 225179981368,5504 \quad 98) 944541172034,7904 \cdot 2 = 1889082344069,5808$$

$$99) 225179981368,5504 \cdot 2 = 450359962737,1008 \quad 100) 1889082344069,5808 \cdot 2 = 3778164688139,1616$$

$$101) 450359962737,1008 \cdot 2 = 900719925474,2016 \quad 102) 3778164688139,1616 \cdot 2 = 7556329376278,3232$$

$$103) 900719925474,2016 \cdot 2 = 1801439850948,4032 \quad 104) 7556329376278,3232 \cdot 2 = 15112658752556,6464$$

$$105) 1801439850948,4032 \cdot 2 = 3602879701896,8064 \quad 106) 15112658752556,6464 \cdot 2 = 30225317505113,2928$$

$$107) 3602879701896,8064 \cdot 2 = 7205759403793,6128 \quad 108) 30225317505113,2928 \cdot 2 = 60450635010226,5856$$

$$109) 7205759403793,6128 \cdot 2 = 14411518807587,2256 \quad 110) 60450635010226,5856 \cdot 2 = 120901270020453,1712$$

$$111) 14411518807587,2256 \cdot 2 = 28823037615174,4512 \quad 112) 120901270020453,1712 \cdot 2 = 241802540040906,3424$$

$$113) 28823037615174,4512 \cdot 2 = 57646075230348,9024 \quad 114) 241802540040906,3424 \cdot 2 = 483605080081812,6848$$

$$115) 57646075230348,9024 \cdot 2 = 115292150460697,8048 \quad 116) 483605080081812,6848 \cdot 2 = 96721$$

$$D = 2,5 = 000,1000 \rightarrow 1,01.2' \Rightarrow \begin{cases} e = 127 + 1 = 128 = 1000000 \\ s = 0 \\ m = 01000000000000000000 \end{cases}$$

c) punto fijo :

$$\begin{array}{r}
 \begin{array}{r}
 1 & 1 & 1 & 1 \\
 0 & 1 & 1 & 1 \\
 0 & 0 & 1 & 0
 \end{array}
 \begin{array}{r}
 | & & & \\
 | & & & \\
 | & & & \\
 1 & 0 & 1 & 0 \\
 1 & 0 & 0 & 0
 \end{array}
 \begin{array}{r}
 = 1, 1, 6 + 1.9.40^{\circ} \\
 = 2, 5.40^{\circ} \\
 = 60, 125
 \end{array}
 \end{array}$$

punto flotante :

1) Calcular las diferencias entre los exponentes:

$$d = 10 - 0 \mid = 0$$

2) como de posiciones a la derecha le come del mismo menor:

$$2,5.\text{v}^\circ \rightsquigarrow 2,5.\text{w}^\circ$$

3) Em colementos y sumos:

$$\begin{array}{r}
 & 7 & 6 & 7 & 1 & 9 \\
 + & 1 & 5 & 0 & 0 & 0 \\
 \hline
 2 & 1 & 7 & 1 & 9
 \end{array}$$

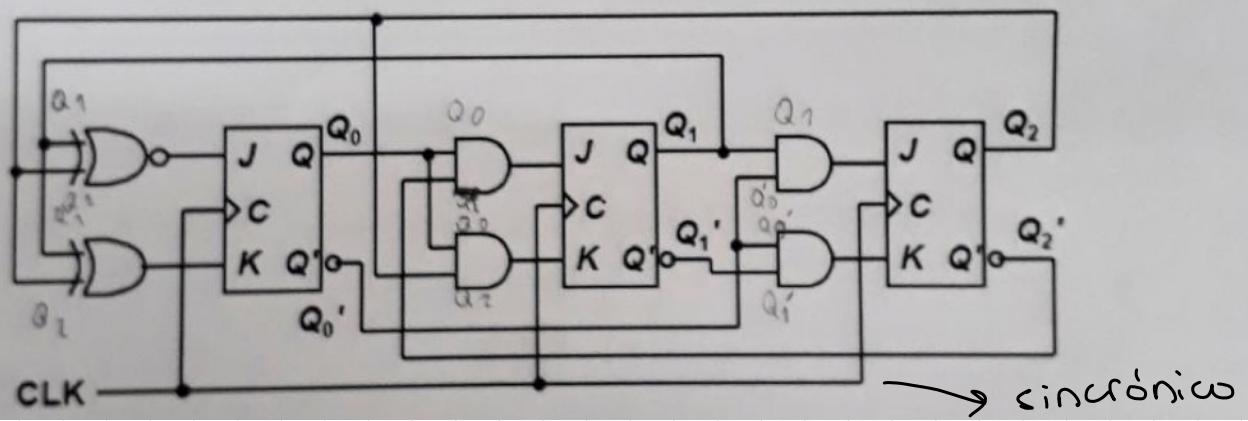
4) El exponente de resultado es el exponente del numero mayor

W, 1719. W°

5) Normalizó:

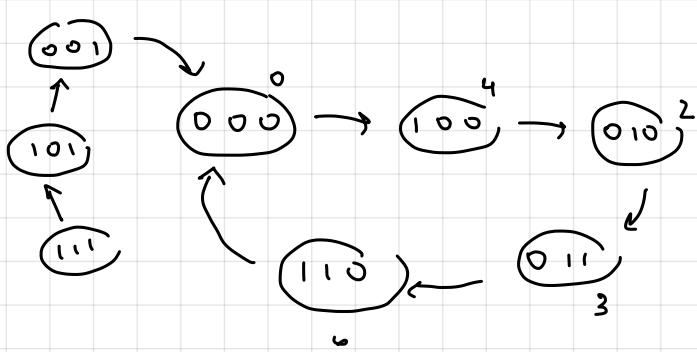
1,01719. w'

3) Analizar el siguiente circuito indicando todos sus estados y transiciones. Explicar si puede ser usado como contador y en ese caso cuál es su módulo y código de cuenta.



	omt.			actual			$J_0 = \bar{Q}_1 \bar{Q}_2 + Q_1 Q_2$
	J_0	K_0	J_1	K_1	J_2	K_2	
1	1	0	0	0	0	0	$Q_0 = 1, Q_1 = 0, Q_2 = 0$
2	1	0	1	0	0	0	$Q_0 = 0, Q_1 = 1, Q_2 = 0$
3	0	1	0	0	1	0	$Q_0 = 0, Q_1 = 1, Q_2 = 1$
4	1	0	0	0	1	1	$Q_0 = 1, Q_1 = 1, Q_2 = 0$
5	0	1	1	0	0	0	$Q_0 = 0, Q_1 = 0, Q_2 = 0$
6	0	1	0	0	0	1	$Q_0 = 0, Q_1 = 0, Q_2 = 1$
7	0	1	0	1	0	0	$Q_0 = 0, Q_1 = 0, Q_2 = 0$
8	1	0	0	1	0	1	$Q_0 = 0, Q_1 = 1, Q_2 = 1$

$$K_2 = \bar{Q}_0 \bar{Q}_1$$



contador sincrónico de módulo 5 con código de cuenta

0, 1, 2, 3, 4.

2018C1



5) Diseñar un contador síncrono con capacidad de contar hasta 6 pulsos con un código de cuenta selectable entre binario descendente y binario ascendente a través de una entrada de control en 0 o 1 respectivamente.

CONTADOR SÍNCRONO, módulo 6 \Rightarrow uso 3 FF. Elijo JK.

Armo la secuencia de estados:

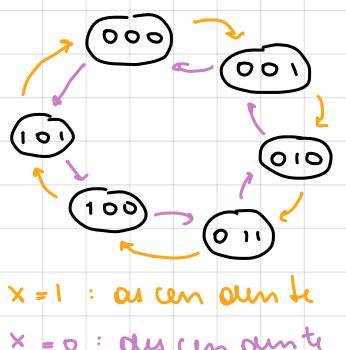


tabla de transiciones:

J	K	Q'	J	K	Q'
0	0	Q	0	0	x
0	1	0	0	1	x
1	0	1	1	0	x
1	1	0	1	1	0

\Rightarrow

\Rightarrow

\Rightarrow

Armo la tabla de verdad:

x	Q_0, Q_1, Q_2	Q_0', Q_1', Q_2'	J_0, K_0	J_1, K_1	J_2, K_2
0	0 0 0	0 0 1	0 x	0 x	1 x
0	0 0 1	0 1 0	0 x	1 x	x 1
0	0 1 0	0 1 1	0 x	x 0	1 x
0	0 1 1	1 0 0	1 x	x 1	x 1
0	1 0 0	1 0 1	x 0	0 x	1 x
0	1 0 1	1 1 0	x 0	1 x	x 1
0	1 1 0	x x x	x x	x x	x x
0	1 1 1	x x x	x x	x x	x x
1	0 0 0	1 0 1	1 x	0 x	1 x
1	0 0 1	0 0 0	0 x	0 x	x 1
1	0 1 0	0 0 1	0 x	x 1	1 x
1	0 1 1	0 1 0	0 x	x 0	x 1
1	1 0 0	0 1 1	x 1	1 x	1 x
1	1 0 1	1 0 0	x 0	0 x	x 1
1	1 1 0	x x x	x x	x x	x x
1	1 1 1	x x x	x x	x x	x x

simplifíco con mapas de karnaugh:

J_0	Q_0, Q_1, Q_2	Q_0', Q_1', Q_2'
x \bar{Q}_0, \bar{Q}_2	00 01 11 10	00 0 0 1 0
00	0 0 0	R R R R
01	R R R R	0 0 0 R R
11	R R R R	1 0 R R R
10	1 0 0 0	R R R R

J_1	Q_0, Q_1, Q_2	Q_0', Q_1', Q_2'
x \bar{Q}_0, \bar{Q}_2	00 01 11 10	00 0 1 R R
00	0 0 0	R R R R
01	0 0 1	1 0 R R R
11	1 0 1	0 R R R
10	R R R R	R R R R

J_2	Q_0, Q_1, Q_2	Q_0', Q_1', Q_2'
x \bar{Q}_0, \bar{Q}_2	00 01 11 10	00 0 1 R 0
00	R R R R	R R R R
01	R R R R	1 0 R R R
11	R R R R	R R R R
10	R R R R	R 0 R R

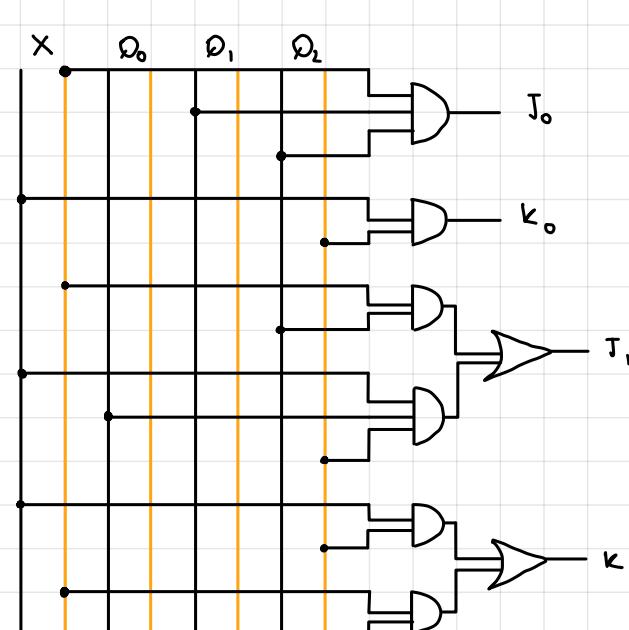
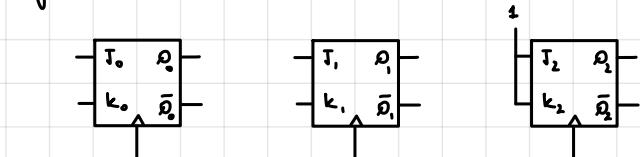
$$J_0 = \bar{x} Q_1 Q_2$$

$$K_0 = x \bar{Q}_2$$

$$J_1 = \bar{x} Q_2 + x Q_0 \bar{Q}_2$$

$$K_1 = x \bar{Q}_2 + \bar{x} Q_2$$

grafico el circuito:



2022 C1

1) Dados los siguientes números de 8 bits

$$A = 10101100 \quad B = 11010111 \quad C = 56h \quad D = 65h$$

(a) obtener $A+B$ $C+D$ considerando que son enteros sin signo

(b) obtener $A+B$ $C+D$ considerando que son enteros en complemento a 2

(c) obtener $A+B$ $C+D$ como en punto fijo con igual cantidad de dígitos de p. entera y fracc

(d) obtener $A+(-B)$ $C+(-D)$ ~~considerando que son enteros sin signo~~

En todos los casos dar valores en decimal, indicar flags y si el resultado es representable o no justificando.

a) Enteros sin signo:

$$\begin{array}{r} \begin{array}{r} 1 & 1 & 1 & 1 & 1 \\ + & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ \hline 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{array} \end{array} \quad \begin{array}{l} C = 1 \\ V = 0 \end{array} \quad \begin{array}{l} Z = 0 \\ N = 0 \end{array} \quad \begin{array}{l} P = 0 \end{array}$$

$$A = 10101100 = 4 + 8 + 32 + 128 = 172$$

128 64 32 16 8 4 2 1

$$B = 11010111 = 1 + 2 + 4 + 16 + 64 + 128 = 215$$

128 64 32 16 8 4 2 1

$$\left. \begin{array}{r} 1 & 1 & 2 \\ + & 2 & 1 & 5 \\ \hline 3 & 8 & 7 \end{array} \right\} > 255 \Rightarrow \text{se va del rango}$$

y esto bien que $C=1$.

• $C+D$:

$$\begin{array}{r} \begin{array}{r} 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ + & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ \hline 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \end{array} \end{array} \quad \begin{array}{l} C = 0 \\ V = 0 \end{array} \quad \begin{array}{l} Z = 0 \\ N = 0 \end{array} \quad \begin{array}{l} P = 1 \end{array}$$

$$\left. \begin{array}{r} 1 & 0 & 1 \\ + & 8 & 6 \\ \hline 1 & 8 & 7 \end{array} \right\} < 255 \Rightarrow \text{no se va del rango} \therefore C = 0$$

b) Enteros en Cad:

• $A+B$:

Busco A

$$\begin{array}{r} 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ + & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ \hline 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \end{array} \rightarrow 84 \Rightarrow A = -84$$

Busco B

$$\begin{array}{r} 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ + & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \end{array} \rightarrow 41 \Rightarrow B = -41$$

$$\left. \begin{array}{r} -84 \\ -41 \\ \hline -125 \end{array} \right\} \rightarrow \text{Esto} \quad -128 < -125 < 127$$

$$\Rightarrow V = 0$$

Busco $A+B$

$$\begin{array}{r} 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ + & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ \hline 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \end{array} \quad \begin{array}{l} C = 1 \\ V = 0 \end{array} \quad \begin{array}{l} Z = 0 \\ N = 1 \end{array} \quad \begin{array}{l} P = 0 \end{array}$$

como no se prende, la suma es representable

• $C+D$:

$$\begin{array}{r} \begin{array}{r} 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ + & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ \hline 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \end{array} \end{array} \quad \begin{array}{l} C = 0 \\ V = 1 \end{array} \quad \begin{array}{l} Z = 0 \\ N = 1 \end{array} \quad \begin{array}{l} P = 1 \end{array} \quad \left. \begin{array}{l} \text{Como se prende el overflow la suma} \\ \text{se va del rango Veremos que } \oplus + \ominus = \ominus \end{array} \right\}$$

C y D son positivos \Rightarrow en Cad quedan igual.

c) Punto fijo:

$$\begin{array}{r} \begin{array}{r} 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ + & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ \hline 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \end{array} \end{array} \quad \begin{array}{l} 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ + & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ \hline 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \end{array} \quad \begin{array}{l} \rightarrow 10, (2^{-1} + 2^{-2}) \\ \rightarrow 13, (2^{-2} + 2^{-3} + 2^{-4}) \\ \rightarrow 8, (2^{-3} + 2^{-4}) \end{array} \quad \begin{array}{l} C = 1 \\ V = 0 \end{array} \quad \begin{array}{l} N = 0 \\ Z = 0 \end{array} \quad \begin{array}{l} P = 0 \end{array}$$

Como son números sin signo y se prendió el carry, la suma se va del rango.

$$\bullet C = 0 \ 1 \ 0 \ 1, 0 \ 1 \ 1 \ 0 \quad \left. \begin{array}{l} \\ \\ \end{array} \right\} + \begin{array}{r} 1 \\ 0 \ 1 \ 0 \ 1, 0 \ 1 \ 1 \ 0 \rightarrow 5, 6/16 \\ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 0 \ 1 \rightarrow 6, 5/16 \\ \hline 1 \ 0 \ 1 \ 1, 1 \ 0 \ 1 \ 1 \rightarrow 11, 11/16 \end{array} \quad \begin{array}{l} C = \emptyset \\ V = \emptyset \\ \Sigma = \emptyset \\ P = 1 \end{array}$$

Como son números con signo y no se premió el carry, la suma se tuvo dentro del rango.

$$d) \bullet 4 + (-3) \equiv 4 + \text{Ca2}(3) : + \begin{array}{r} 1 \\ 0 \ 1 \ 0 \ 1, 0 \ 1 \ 1 \ 0 \ 0 \\ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1 \\ \hline 1 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \end{array} \xrightarrow{\text{verifico}} \begin{array}{r} 1 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \\ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \\ \hline 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 1 \end{array} \rightarrow 43 \Rightarrow 4 + \text{Ca2}(3) = 43$$

$C = \emptyset \quad \Sigma = \emptyset$
 $V = \emptyset \quad P = \emptyset$
 $N = 1$

Como $V = \emptyset$ la suma no se va del rango.

$$\bullet C + (-D) \equiv C + \text{Ca2}(D) : + \begin{array}{r} 1 \ 1 \ 1 \ 1 \\ 0 \ 1 \ 0 \ 1, 0 \ 1 \ 1 \ 0 \\ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 1 \ 1 \\ \hline 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \end{array} \xrightarrow{\text{verifico}} \begin{array}{r} 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \\ 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 0 \\ \hline 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \end{array} \rightarrow 15 \Rightarrow C + \text{Ca2}(D) = 15$$

$\text{Ca2}(D) : 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 0 \ 1$
 $\sim \sim + 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0$
 $\hline 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 1 \ 1$

$C = \emptyset \quad \Sigma = \emptyset$
 $V = \emptyset \quad P = \emptyset$
 $N = 1$

Como $V = \emptyset$ la suma no se va del rango.

2) Dado el número real $X = 11.375$

- (a) indicar su representación como punto flotante precisión simple normalizado (bit a bit y en hexadecimal)
 (b) indicar las representaciones normalizadas de $(4*X)$ y de $(-1*X/8)$
 (c) representar X en punto fijo de 32 bits dedicando 16 bits a la parte fraccionaria
 (d) Explicar ventajas y desventajas comparativas entre punto fijo y punto flotante

a) Punto Flotante: simple Precisión ($s=1$, $e=8$, $m=23$)

$$11 \frac{1}{10} = 1101.1$$

$$\begin{aligned} 0,375 \cdot 2 &= 0,750 \\ 0,750 \cdot 2 &= 1,5 \\ 0,5 \cdot 2 &= 1,0 \\ 0,0 \cdot 2 &= 0,0 \end{aligned}$$

$$\Rightarrow 0,375 \frac{1}{10} = 0,0110_2$$

luego:

$$11,375 \frac{1}{10} = 1101,0110 \xrightarrow{\text{normalizado}} 1,1010110 \cdot 2^3 \rightarrow e = 12 + 3 = 130$$

$$l = 10000010$$

128 64 32 16 8 4 2 1

$$m = \underbrace{1010110}_{7} 0000000000000000000000$$

finalmente $11,375$ en pto. flotante es:

$$0100|0001|0101|0110|0000|0000|0000|0000 = 4156\ 0000\ h$$

4 1 5 6

$$b) 4x = 2^2 \cdot 1,1010110 \cdot 2^3 = 1,1010110 \cdot 2^5$$

$$-\frac{x}{8} = -x \cdot 2^{-3} = -1,1010110 \cdot 2^3 \cdot 2^{-3} = -1,1010110$$

c) $1101,0110$ en 32 bits con 16 para la parte fraccionaria.

yo use 4 bits para la parte entera y 4 para la fraccionaria \Rightarrow relleno con 12 bits adelante y otros:

$$0000000000001101,0110000000000000$$

d) En Punto fijo tengo que, valgo la redundancia, fijar la cantidad de bits que voy a destinar tanto para la parte entera como para la fraccionaria. Mientras que, en Punto flotante puedo representar tantos decimales como me queden disponibles en la mantisa luego de colocar la parte entera.

Es decir, si estoy trabajando en 32 bits y destino 16 para la parte entera en P. fijo, me quedan 16 bits para la parte fraccionaria y tendré una cota de error de 2^{-17} . Ahora bien, en Punto flotante podré representar como máximo hasta 23 decimales teniendo así una cota de error de 2^{-24} .

Ejemplo:

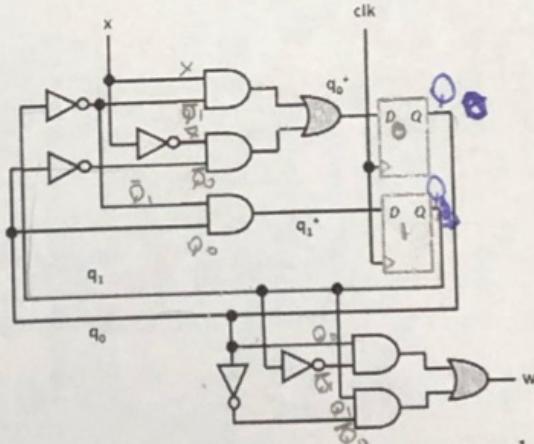
$$\text{Si } A = 1,110010011101001000111010$$

$$\text{en P. fijo: } 0000000000000001,1100100111010010$$

$$\text{en P. flot. } 1,1100100111010010001110$$

represento más decimales

5) Dado el siguiente circuito determinar su secuencia de estados considerando la entrada X y determinando la salida W de cada uno de ellos



• E CONTADOR

• Cómo se incorpora
W al contador?

$$D_0 = X \bar{Q}_1 + \bar{X} \bar{Q}_0$$

$$D_1 = \bar{Q}_1 Q_0$$

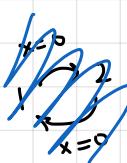
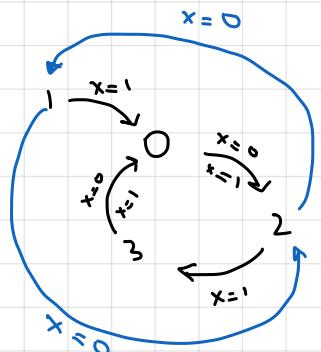
$$W = Q_0 \bar{Q}_1 + \bar{Q}_1 \bar{Q}_0$$

$$X \quad W \quad \{ Q_0 \quad Q_1 \} \quad \{ \bar{Q}_0 \quad \bar{Q}_1 \} \quad D_0 \quad D_1$$

1	0	0	0	1	0	1	0
1	1	1	0	1	1	1	1
1	0	1	1	0	0	0	0
1	1	0	1	0	0	0	0
0	0	0	0	1	0	1	0
0	1	1	0	0	1	0	1
0	1	0	1	1	0	1	0
0	0	1	0	0	0	0	0

$$x=1 \quad \left\{ \begin{array}{l} 0 \ 0 \rightarrow 1 \ 0 \Rightarrow 0 \rightarrow 2 \\ 1 \ 0 \rightarrow 1 \ 1 \Rightarrow 2 \rightarrow 3 \\ 1 \ 1 \rightarrow 0 \ 0 \Rightarrow 3 \rightarrow 0 \\ 0 \ 1 \rightarrow 0 \ 0 \Rightarrow 1 \rightarrow 0 \end{array} \right.$$

$$x=0 \quad \left\{ \begin{array}{l} 0 \ 0 \rightarrow 1 \ 0 \Rightarrow 0 \rightarrow 2 \\ 1 \ 0 \rightarrow 0 \ 1 \Rightarrow 2 \rightarrow 1 \\ 0 \ 1 \rightarrow 1 \ 0 \Rightarrow 1 \rightarrow 2 \\ 1 \ 1 \rightarrow 0 \ 0 \Rightarrow 3 \rightarrow 0 \end{array} \right.$$



6) Utilizando tres flipflops tipo JK diseña un circuito que tiene una entrada de control que en caso de estar en 0 a la salida del circuito se observará la siguiente secuencia iterativa:

100 - 010 - 001 - 100

y que en caso de que la entrada de control esté en 1 la secuencia será:

001 - 010 - 100 - 001

El cambio de una salida a la siguiente se produce toda vez que ocurre un flanco descendente en la entrada de pulsos a contar, la cual está conectada sincrónicamente a los tres flipflop

Diseño de un contador sincrónico:

M	Q_0	Q_1	Q_2	Q_0^*	Q_1^*	Q_2^*	J_0	K_0	J_1	K_1	J_2	K_2
0	1	0	0	0	1	0	x	1	1	x	0	x
0	0	1	0	0	0	1	0	x	x	1	x	
0	0	0	1	1	0	0	1	x	0	x	x	1
0	0	0	0	x	x	x	x	x	x	x	x	x
0	0	1	1	x	x	x	x	x	x	x	x	x
0	1	0	1	x	x	x	x	x	x	x	x	x
0	1	1	0	x	x	x	x	x	x	x	x	x
0	1	1	1	x	x	x	x	x	x	x	x	x
1	0	0	1	0	1	0	0	x	1	x	x	1
1	0	1	0	1	0	0	1	x	1	0	x	
1	1	0	0	0	0	1	x	1	0	x	1	x
1	0	0	0	x	x	x	x	x	x	x	x	x
1	0	1	1	x	x	x	x	x	x	x	x	x
1	1	0	1	x	x	x	x	x	x	x	x	x
1	1	1	1	x	x	x	x	x	x	x	x	x

Tabla de transiciones:

J	K	Q^*	Q	J	K	Q^*	Q
0	0	0	0	0	0	0	0
0	1	0	0	0	1	0	1
1	0	1	0	1	0	1	0
1	1	1	0	1	1	0	0

Simplificación con Karnaugh:

J_0	Q_1	Q_2	00	01	11	10
M_{Q_2}	Q_1	Q_2	R	1	R	0
00	R	R	R	R	R	R
01	R	R	R	R	R	R
11	R	R	R	R	R	1
10	R	0	R	R	R	R

J_1	Q_1	Q_2	00	01	11	10
M_{Q_2}	Q_1	Q_2	R	R	R	R
00	R	R	R	R	R	R
01	1	R	R	R	R	R
11	1	R	R	R	R	R
10	R	R	R	R	R	R

J_2	Q_1	Q_2	00	01	11	10
M_{Q_2}	Q_1	Q_2	R	0	R	R
00	R	R	R	R	R	R
01	1	R	R	R	R	R
11	0	R	R	R	R	R
10	R	1	R	R	R	R

J_0	Q_1	Q_2	00	01	11	10
M_{Q_2}	Q_1	Q_2	R	R	R	1
00	R	R	R	R	R	1
01	R	R	R	R	R	R
11	R	R	R	R	R	R
10	R	R	R	R	R	1

$$J_0 = \bar{M} Q_2 + M Q_0$$

$$K_0 = 1$$

$$J_1 = \bar{M} Q_0 + M \bar{Q}_0$$

$$K_1 = 1$$

J_2	Q_1	Q_2	00	01	11	10
M_{Q_2}	Q_1	Q_2	R	R	R	1
00	R	R	R	R	R	1
01	0	R	R	R	R	1
11	1	R	R	R	R	1
10	R	R	R	R	R	0

J_0	Q_1	Q_2	00	01	11	10
M_{Q_2}	Q_1	Q_2	R	1	R	R
00	R	R	R	1	R	R
01	R	R	R	R	R	R
11	R	R	R	R	R	R
10	R	1	R	R	R	R

$$J_2 = \bar{M} \bar{Q}_0 + M Q_0$$

$$K_2 = 1$$

luego:

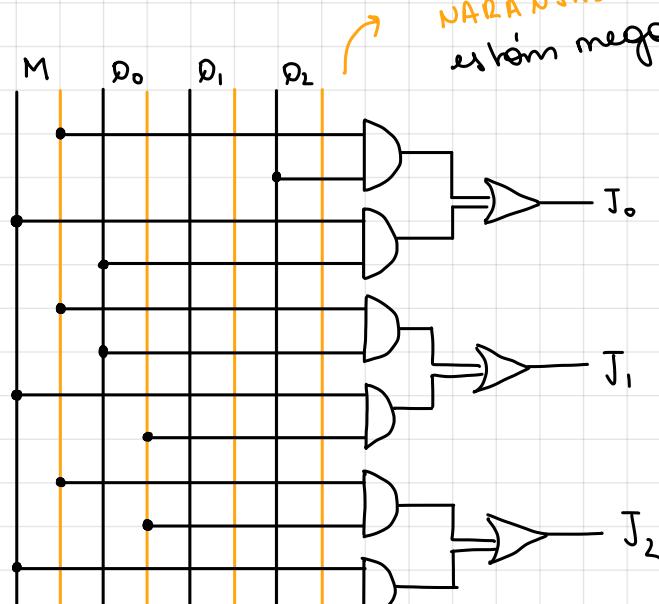
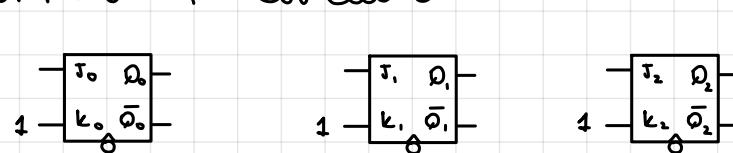
$$J_0 = \bar{M} Q_2 + M Q_0 \quad K_0 = 1$$

$$J_1 = \bar{M} Q_0 + M \bar{Q}_0 \quad K_1 = 1$$

$$J_2 = \bar{M} \bar{Q}_0 + M Q_0 \quad K_2 = 1$$

los cables
NARANJAS
están negados

Forma el circuito:

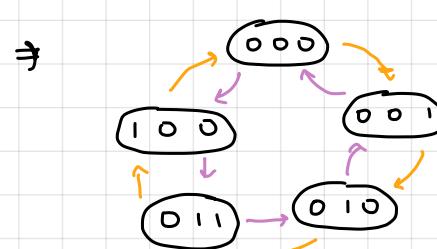


- Diseñar un contador sincrónico binario que permite contar a elección de forma ascendente o descendente. Debe ser de módulo 5 (rango 0, 1, 2, 3, 4) y estar implementado con un FF-JK.

Como el módulo del contador es 5 necesita 3 bits para almacenar los datos, luego diseno la secuencia:

0)	0	0	0
1)	0	0	1
2)	0	1	0
3)	0	1	1
4)	1	0	0
5)	1	0	1
6)	1	1	0
7)	1	1	1

redundancia
pues es enunciado
no me pide modo
también estos estados



ascendente $\Rightarrow x = 1$

descendente $\Rightarrow x = 0$

Construyo la tabla de verdad:

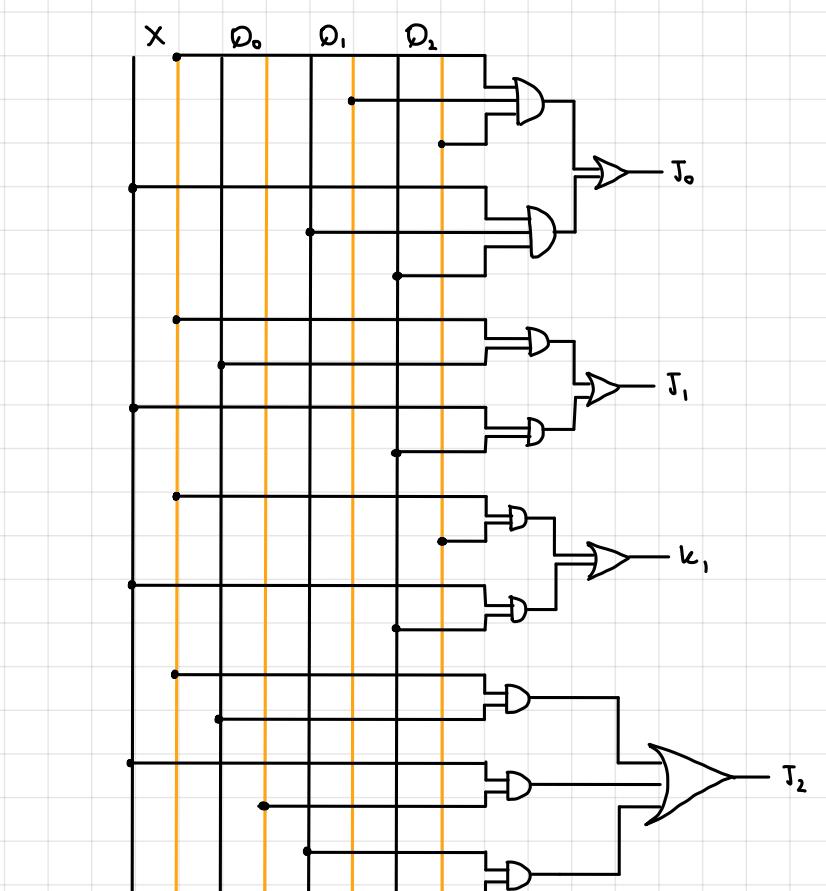
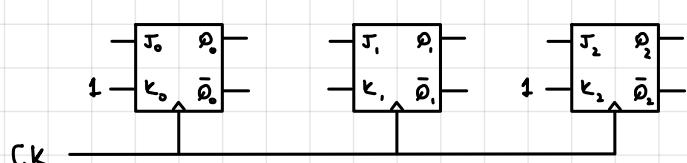
x	Q_0	Q_1	Q_2	\bar{Q}_0	\bar{Q}_1	\bar{Q}_2	J_0	K_0	J_1	K_1	J_2	K_2	\bar{P}_0	\bar{P}_1	\bar{P}_2	T	K
1	0	0	0	0	0	1	0	x	0	x	1	x	0	0	0	x	1
1	0	0	1	0	1	0	0	x	1	x	x	1	0	1	1	x	0
1	0	1	0	0	1	1	0	x	x	0	x	1	0	x	1	x	1
1	0	1	1	1	0	0	1	x	x	1	x	1	1	x	0	x	0
1	1	0	0	0	0	0	x	1	0	x	0	x	0	0	0	x	1
1	1	0	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x
1	1	1	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x
1	1	1	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x
0	0	0	0	1	0	0	1	x	0	x	0	x	0	0	0	x	1
0	0	0	1	0	0	0	0	x	0	x	0	x	x	1	1	x	0
0	0	1	0	0	0	1	0	x	x	1	x	1	0	x	1	x	1
0	0	1	1	0	1	0	0	x	x	0	x	x	1	x	1	x	1
0	1	0	0	0	1	1	x	1	1	x	1	x	1	0	x	1	x
0	1	0	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x
0	1	1	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x
0	1	1	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Simplifico con mapas de karnaugh:

J_0	Q_0	Q_1	Q_2	\bar{Q}_0	\bar{Q}_1	\bar{Q}_2	00	01	11	10
x	0	0	0	1	0	0	0	0	0	0
0	0	1	0	R	R	R	R	R	R	R
0	1	R	R	R	R	R	R	R	R	R
1	R	R	R	R	R	R	R	R	R	R
1	0	0	0	1	0	0	0	0	0	0

J_1	Q_0	Q_1	Q_2	\bar{Q}_0	\bar{Q}_1	\bar{Q}_2	00	01	11	10
x	0	0	0	0	0	0	R	R	R	R
0	0	1	0	0	x	0	R	R	R	R
0	1	R	R	R	R	R	R	R	R	R
1	R	R	R	R	R	R	R	R	R	R
1	0	0	0	1	0	0	0	0	0	0

Finalmente, grafico el circuito:



K_1	Q_0	Q_1	Q_2	\bar{Q}_0	\bar{Q}_1	\bar{Q}_2	00	01	11	10
x	R	R	0	1	1	1	R	R	R	R
0	1	R	R	R	R	R	R	R	R	R
0	1	R	R	R	R	R	R	R	R	R
1	R	R	R	R	R	R	R	R	R	R
1	0	R	R	1	0	0	0	0	0	0

J_2	Q_0	Q_1	Q_2	\bar{Q}_0	\bar{Q}_1	\bar{Q}_2	00	01	11	10
x	0	0	0	0	0	0	R	R	R	R
0	0	1	0	0	x	0	R	R	R	R
0	1	R	R	R	R	R	R	R	R	R
1	R	R	R	R	R	R	R	R	R	R
1	0	1	0	1	R	R	R	R	R	R

$$K_1 = \bar{x} \bar{Q}_2 + x \bar{Q}_2$$

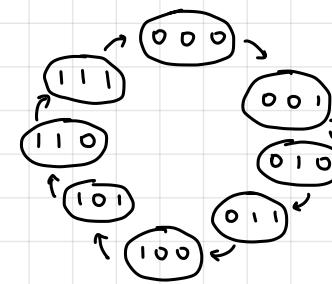
$$J_2 = \bar{x} \bar{Q}_0 + x \bar{Q}_0 + \bar{Q}_1 \bar{Q}_2$$

• Diseñar un contador sincrónico de doble binario de módulo 8 con ff-T.

"módulo 8" \Rightarrow uno 3 ff. Armo la secuencia del contador.

Armo la tabla de verdad:

Q_0	Q_1	Q_2	Q_0^+	Q_1^+	Q_2^+	T_0	T_1	T_2
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	0	1	0	1	1
1	1	0	1	1	1	0	0	1
1	1	1	0	0	0	1	1	1



Se implemico con mapas de karnaugh:

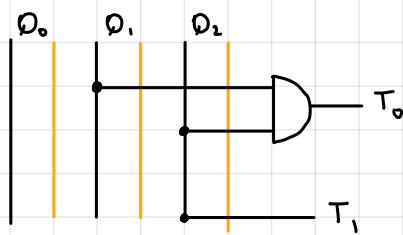
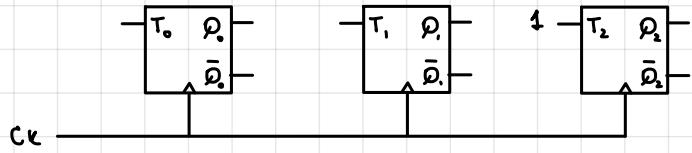
T_0	Q_1	Q_2	00	01	11	10
0	0	0	0	0	1	0
1	0	0	1	0	0	0

T_1	Q_1	Q_2	00	01	11	10
0	0	0	0	0	1	0
1	0	0	1	1	1	0

$$T_0 = Q_1 Q_2$$

$$T_1 = Q_2$$

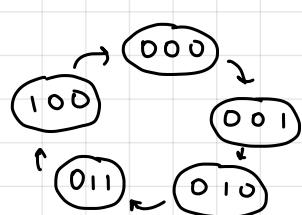
Finalmente, grafico el circuito:



• Dibujar un contador binario asincrónico que sea de modulo 5 al final de cada conteo.

"modulo 5" \Rightarrow uno 3 ff. El tipo τ_k , $t=k=1$ (tipo Toggle)

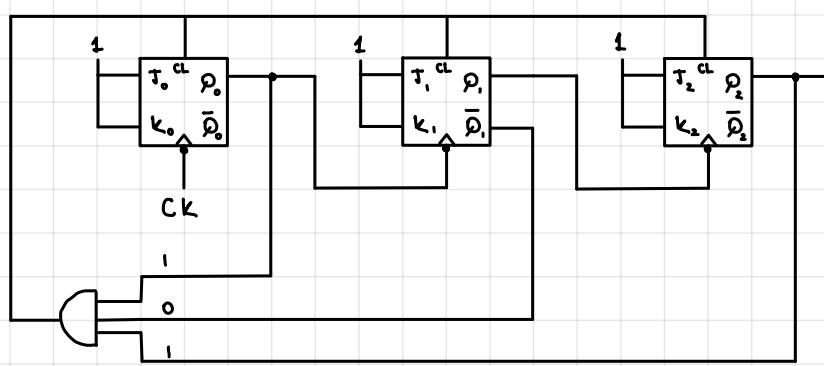
Armo la secuencia de estados:



Estados prohibidos:

5) 1 0 1
6) 1 1 0
7) 1 1 1

Gráfico del circuito:



RECU

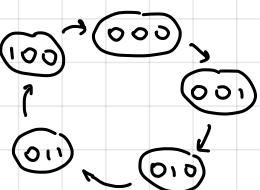


- Construir un contador de módulo 5 ($0, 1, 2, 3, 4$). Del estado 1 al 3 enciende un LED.

SÍNCRONICO

módulo 5 $\Rightarrow 2^3 = 8 > 5 \rightarrow$ uso 3 FF's JK.

Defino los estados:



Construyo la tabla de verdad:

Q_0, Q_1, Q_2	D_0^*, D_1^*, D_2^*	J_0, K_0	J_1, K_1	J_2, K_2	Q
0 0 0	0 0 1	0 x	0 x	1 x	0 0 0 x
0 0 1	0 1 0	0 x	1 x	x 1	0 1 1 x
0 1 0	0 1 1	0 x	x 0	1 x	1 0 x 1
0 1 1	1 0 0	1 x	x 1	x 1	1 1 x 0
1 0 0	0 0 0	x 1	0 x	0 x	
1 0 1	0 1 0	0 1	1 1	0 1	
1 1 0	0 1 0	0 1	0 0	0 1	
1 1 1	0 0 0	1 1	1 1	0 1	

estados prohibidos

J	K	Q
0	0	0
0	1	0
1	0	1
1	1	1

J_0	Q, \bar{Q}_2	D_0	00	01	11	10
0	0 0 0	0	0	0	1	0
1	x x x	x	x	x	x	x

J_1	Q, \bar{Q}_2	D_0	00	01	11	10
0	0 0	0	0	1	x	x
1	0	x	x	x	x	x

J_2	Q, \bar{Q}_2	D_0	00	01	11	10
0	x	0	x	1	0	0
1	x	x	x	x	x	x

J_2	Q, \bar{Q}_2	D_0	00	01	11	10
0	1	x	x	x	1	0
1	0	x	x	x	x	x

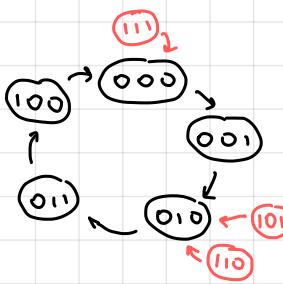
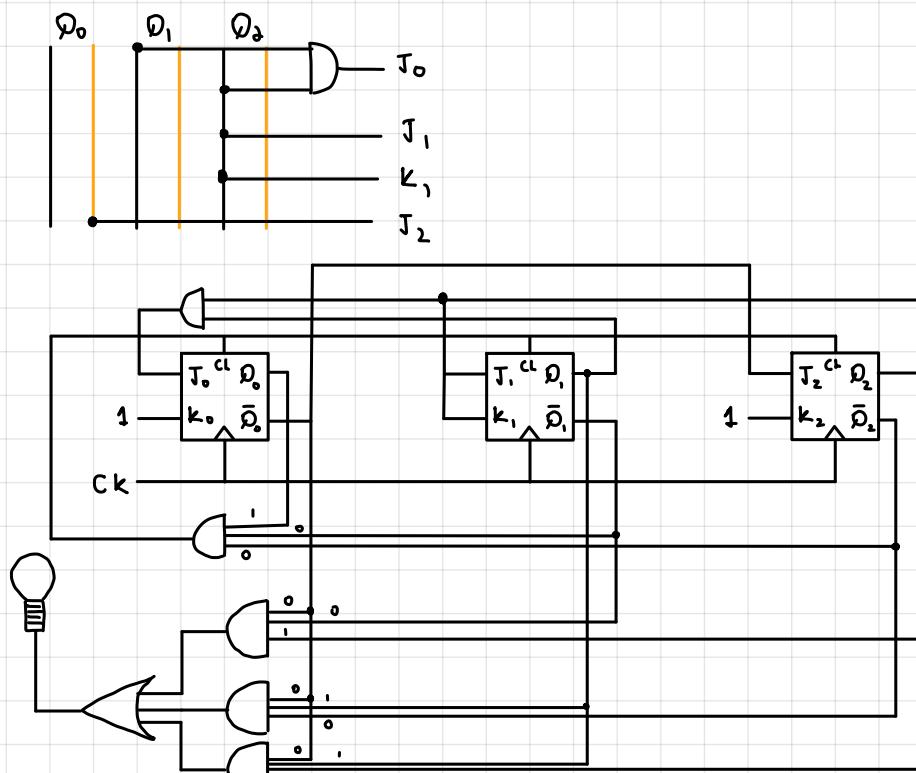
$$J_0 = Q_1 Q_2$$

$$J_1 = Q_2$$

$$K_1 = Q_2$$

$$J_2 = \bar{Q}_0$$

Construyo el circuito.



• prendo el led en los estados 1, 2 y 3. En otro caso, se apaga.

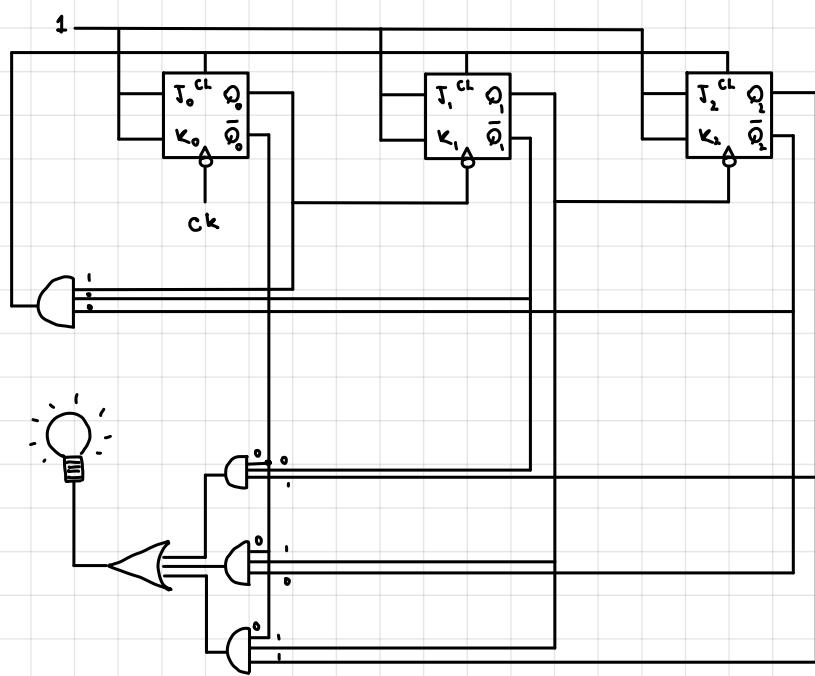
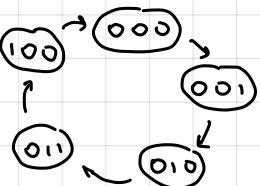
• limpia el contador en el estado 4.

- Construir un contador de módulo 5 (0, 1, 2, 3, 4). Del estado 1 al 3 enciende un LED.

ASÍNCRONICO

módulo 5 $\Rightarrow 2^3 = 8 > 5 \rightarrow$ uso 3 FF's JK.

Definir los estados:



- Inicio el contador en el estado 4.
- Puede encender en los estados 1, 2 y 3. En otros casos, se apaga.

5) Sobre la base de flip-flops JK diseñar un contador sincrónico con capacidad de contabilizar hasta 4 pulsos y cuya salida representa la cuenta segun:

- 0 pulsos... 0
- 1 pulso ... 1
- 2 pulsos... 3
- 3 pulsos... 5
- 4 pulsos... 7

0 → 0	}
1 → 1	
2 → 3	
3 → 5	
4 → 7	

U10 3 ff-JK

Construyo la tabla de verdad:

Q_0	Q_1	Q_2	Q_0^*	Q_1^*	Q_2^*	J_0	K_0	J_1	K_1	J_2	K_2
0	0	0	0	0	0	0	x	0	x	0	x
0	0	1	0	0	1	0	x	0	x	0	x
0	1	0	0	1	1	0	x	x	0	1	x
0	1	1	1	0	1	1	x	x	1	x	1
1	0	0	1	1	1	x	0	1	x	1	x
1	0	1	x	x	x	x	x	x	x	x	x
1	1	0	x	x	x	x	x	x	x	x	x
1	1	1	x	x	x	x	x	x	x	x	x

estados prohibidos

Simplifico con mapas de karnaugh:

J_0	Q_1	Q_2	Q_0	Q_1^*	Q_2^*
0	0	0	0	1	0
1	x	x	x	x	x

J_0	Q_1	Q_2	Q_0	Q_1^*	Q_2^*
0	x	x	x	x	x
1	0	x	x	x	x

J_1	Q_1	Q_2	Q_0	Q_1^*	Q_2^*
0	0	0	0	x	x
1	1	x	x	x	x

J_2	Q_1	Q_2	Q_0	Q_1^*	Q_2^*
0	x	x	1	0	0
1	x	x	x	x	x

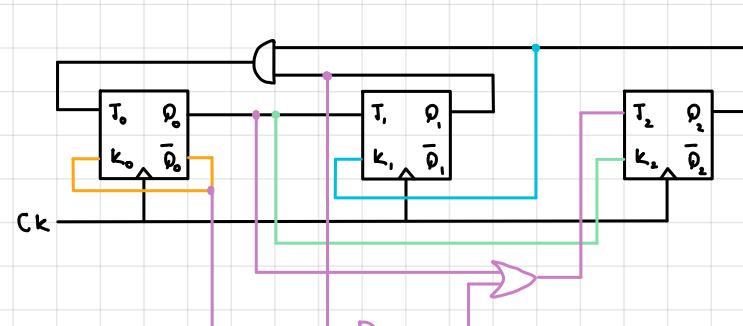
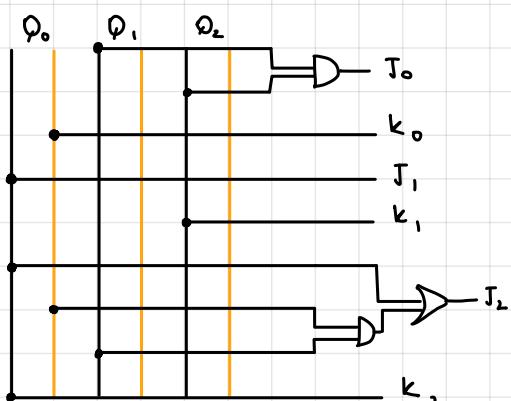
J_2	Q_1	Q_2	Q_0	Q_1^*	Q_2^*
0	0	x	x	x	1
1	1	x	x	x	x

J_2	Q_1	Q_2	Q_0	Q_1^*	Q_2^*
0	x	0	0	0	x
1	x	x	x	x	x

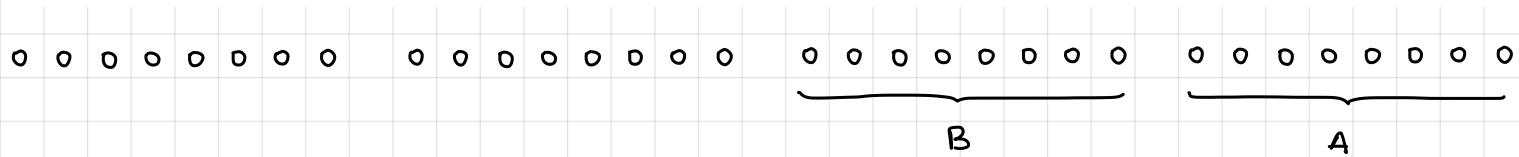
$$J_0 = Q_1 Q_2 \quad J_1 = Q_0 \quad J_2 = Q_0 + \bar{Q}_0 Q_1$$

$$K_0 = \bar{Q}_0 \quad K_1 = Q_2 \quad K_2 = Q_0$$

Construyo el circuito:



1-Se reciben dos números signados de 8 bits por el registro %r7, A=A7..A0 y B= B15..B8. Escribir un programa de ensamblador que compare A y B de manera que retorne 1 si A>B, 0 si son iguales y -1 si A<B.



.begin

.org 2048

sl1 %r7, 24, %r10 ! r10 ← A A A A 0 0 0 0 0 0 0 0 0 0 0 0

sra %r10, 24, %r10 ! r10 ← x x x x x x x x x x x x A A A A

sl1 %r7, 16, %r11 ! r11 ← B B B B 4 4 4 4 0 0 0 0 0 0 0 0

sl2 %r11, 24, %r11 ! r11 ← x x x x x x x x x x x x B B B B

subcc %r10, %r11, %r0 ! r0 ← A - B

bpos uno ! A > B

bneg menos_uno ! A < B

be cero ! A == B

jmp1 %r15, 4, %r0 ! vuelvo

uno: addcc %r0, 1, %r3 ! r3 ← 0 + 1

menos_uno: addcc %r0, -1, %r3 ! r3 ← 0 - 1

cero: st %r3, %r0 ! r3 = r0

.end

2-Lo/la contratan de una empresa para desarrollar un compilador C para procesadores de 32 bits. Realice una buena implementación en ensamblador para las sentencias while, do while, switch y for. Recuerde que estas subrutinas serán llamadas desde un programa principal.

!! while

! while (cond) { code; } ;

while : ba cond

if-true:

cond : and cc '1. f1, 1. f1, '1. f0 ! f1 es la condicion

cond: and cc .11, .11, .10 .11 is a condition

one if true

!! do while

! do { code; } while (cond);

while : ba code

code :

! aca va el codigo

if-true: andcc 1.1, 1.1, 1.10

! F1 es la condicion

bne code

!! for

```
! for (set, cond, update) { code };
```

for : ba cond

cond : and cc 'f1, f1, f0

50% 50%

coffee:

!aca va el código

↳ update

update.

'actual:to' la variable

!! switch

! switch (var) { case n : (code); break ;
default : (code); break ; }

.macro BEQUAL R1, R2, WHERE

subcc R1, R2, .L0

be WHERE

.end macro

switch: ld var, %r1

switch-comp: BEQUAL %r1, cond-n, code-cond-n

!...

ba switch-default

code-cond-n : ! condition n code

! if break

ba switch-end

! else

ba switch-comp

! ...

switch-default : ! default case code

end. sethi 0, .L0

3-Dado el siguiente código:

!-----
!This declares an array of size 5 using literal definitions then
!stores it into an array of size 5 using .dwb

```
.begin
.org 2048
a_start .equ 3000
array_size .equ 5

2048) orcc %r0, array_size, %r1 !CLEAR THE FLAGS
2052) sub %r1, 1, %r1
2056) xor %r31, %i31, %g31
2060) sll %r1, 2, %r1
loop:
2068) ld %r1, [a], %r5
2072) st %r5, %r1, [array]
2076) sub %r1, 4, %r1
2080) be done
2084) subcc %r1, 0, %r0
2088) ba loop
done:
halt
3000) .org a_start
array: .dwb array_size → .equ 5
a: 47, -10, 33, -5, 7
3020
.end
!-----
```

(a) Indicar la tabla de símbolos incluyendo relocalización y referencias externas,

(b) Señalar cuales son las líneas de este código que no tendrán representación en RAM cuando el programa sea cargado para su ejecución.

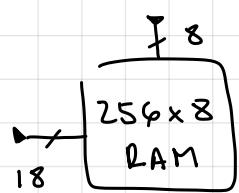
2) symbol	value	global/external	relocatable
a_start	3000	no	no
array_size	5	no	no
a	3020	no	yes
array	3000	no	yes
done	2092	no	yes
loop	2064	no	yes

b) No tiene representación en RAM las directivas que son propias del ensamblador.

1) Un microprocesador puede direccionar 1 Mbyte de RAM, ancho de palabra, 8 bits.

Es decir: bus de address, 20 bits y bus de direcciones 8 bits.

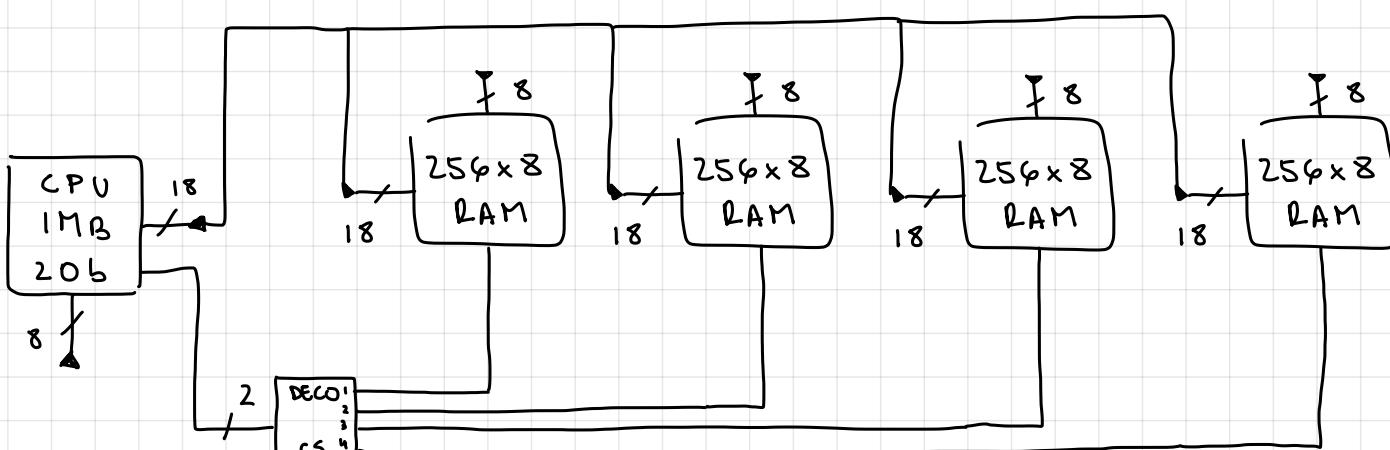
Si se cuenta con una cantidad ilimitada de chips de RAM de 256 kilobytes y de decodificadores, diseñar el mayor mapa de memoria posible para ese microprocesador. Indicar la primera y la última dirección de la RAM que mapea las direcciones más altas.



tengo que expandir
a lo alto.

MAPA de Memoria. $\log_2(18) = 4,17 \Rightarrow$ necesito 4x5 bits

0 0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	MÓDULO 0
3 F F F F	0 0 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	
4 0 0 0 0	0 1 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	
7 F F F F	0 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	
8 0 0 0 0	1 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	MÓDULO 1
B F F F F	1 0 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	
C 0 0 0 0	1 1 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	
F F F F F	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1	
8 bits							



las direcciones más altas son $\underbrace{0 0 0 0}_{\text{primera}}$ y $\underbrace{f f f f f}_{\text{última}}$.

2)

En un procesador ARC el registro Program Counter apunta a la siguiente instrucción guardada en RAM:

ld %r10, %r1, %r7

Detallar los pasos de microprograma que la decodifican y para cada uno de ellos indicar los valores presentes en las entradas y en las salidas de cada uno de los siguientes bloques funcionales:

- * multiplexor de direcciones de la memoria de control
- * incrementador de direcciones de la memoria de control
- * decodificadores de los buses A B y C
- * multiplexores que intervienen en la decodificación de los buses A B y C
- * multiplexor de datos del bus C
- * bus de datos A B y C
- * entrada de direcciones del módulo de memoria RAM

Considere que antes de la ejecución de esta instrucción:

%r10=3000, %r1=8, %r7=1

(Nota: es idéntico al ejercicio 8 del TP7)

ld $\frac{r10}{rs1}, \frac{r1}{rs2}, \frac{r7}{rd}$

Construyo el registro IR:

op = 11

rd = 00111

op3 = 0000000

rs1 = 01010

c = 0

rs2 = 00001

decode = $\frac{111}{op} \frac{00000000}{op3} = 1792_{10}$

luego, se procesa la instrucción 1792 según lo interpretado por el decoder:

1792 || . R[temp0] \leftarrow ADD (R[rs1], R[rs2]); if R[12[3]] $\overset{=0}{\sim}$ THEN GOTO 1794;

A = x

AMUX = 1

B = x

BMUX = 1

C = 100001

CMUX = 0

RD = 0

WR = 0

ALU = 1000

COND = 000

JUMP ADDR = 11100000001

1793 || · $R[ra] \leftarrow \text{AND}(R[\text{temp}0], R[\text{temp}0])$; READ; GOTO 2047;

$A = 100001$

$A_{\text{MUX}} = 0$

$B = 100001$

$B_{\text{MUX}} = 0$

$C = x$

$C_{\text{MUX}} = 1$

$DD = 1$

$WR = 0$

$ALU = 0101$

$COND = 110$

JUMP ADDR = 011111111111

2047 || · $R[pc] \leftarrow \text{INCPC}(R[pc])$; GOTO 0;

$A = 1000000$

$A_{\text{MUX}} = 0$

$B = x$

$B_{\text{MUX}} = x$

$C = 1000000$

$C_{\text{MUX}} = 0$

$DD = 0$

$WR = 0$

$ALU = 1110$

$COND = 110$

JUMP ADDR = 0000000000000000

Luego, se lee la dirección de memoria 3008.

3) Escribir código que recibe a través de %r10 un número en punto flotante y devuelve su valor absoluto (también en punto flotante).

(Nota: es idéntico al ejercicio 15 del TP5)

.begin

.org 2048

lal [comparador], 1.r1

and 1.r10, 1.r1, 1.r10

! pone en 0 el bit más significativo

comparador : + f f f f f f f f

! + f f f f f f f f = 0 111 111 111 111 111 111 111

.end

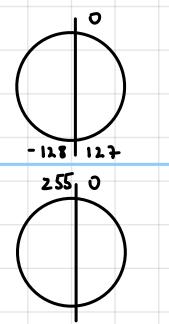
Resumen

TP1 : Sistemas Numéricos

COMPLEMENTO A LA BASE (CA2)

MAYOR: $127_{10} = 0111111_2$ } con signo
 MENOR: $-128_{10} = 100000000_2$

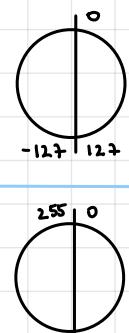
MAYOR: $255_{10} = 1111111_2$ } sin signo
 MENOR: $0_{10} = 0000000_2$



COMPLEMENTO A LA BASE MENOS UNO (CA1)

MAYOR: $127_{10} = 0111111_2$ } con signo
 MENOR: $-127_{10} = 10000000_2$

MAYOR: $255_{10} = 1111111_2$ } sin signo
 MENOR: $0_{10} = 0000000_2$



MAGNITUD Y SIGNO

MAYOR: $127_{10} = 0111111$

MENOR: $-127_{10} = 1111111$

FLAGS

- C (carry): cuando quedó un bit restante después de una operación.
- V (overflow): cuando la suma de dos \oplus da \ominus o dos \ominus da \oplus .
- Z (zero): cuando el resultado es 0.
- N (signo): cuando el resultado es \ominus .
- P (paridad): cuando la cantidad de 1 del numero es par.

PUNTO FLOTANTE

$$\frac{s}{e} \cdot m$$

Simple Precisión: $s = 1, e = 8, m = 23$

Doble Precisión: $s = 1, e = 11, m = 52$

SUMA

- 1) Calculo la diferencia entre los exponentes.
- 2) Corro el punto de los exponentes a la derecha lo como del menor número.
- 3) Encolumno y sumo.
- 4) El exponente del resultado es el exponente del número mayor.
- 5) Normalizar la mantisa.

MULTIPLICACIÓN Y DIVISIÓN

$\times 2$ = desplazar a derecha 1 posición.

$/ 2$ = desplazar a izquierdo 1 posición.

TP 2 : Algèbre ou Boole

COMPUERTAS LÓGICAS

$A \mid Y$		Y	
0 1			
1 0			
$A \mid B \mid Y$		Y	
0 0 0			
0 1 0			
1 0 0			
1 1 1			
$A \mid B \mid Y$		Y	
0 0 1			
0 1 1			
1 0 1			
1 1 0			

TP3 : flip-flops

S	R	Q	Q̄	T	Q(t+1)	Q	D	Q	D	Q(t+1)	Q	J	K	Q(t+1)	
0	0	ant.	0	0	0	0	0	0	0	0	0	0	0	memoria	
0	1	0	1	0	1	0	1	0	1	1	0	0	1	reset	
1	0	1	0	1	0	1	0	0	0	0	1	0	1	set	
1	1	indef	1	1	0	1	1	1	1	1	0	1	1	indefinido	
											1	0	0	1	memoria
											1	0	1	0	reset
											1	1	0	1	set
											1	1	1	0	indefinido

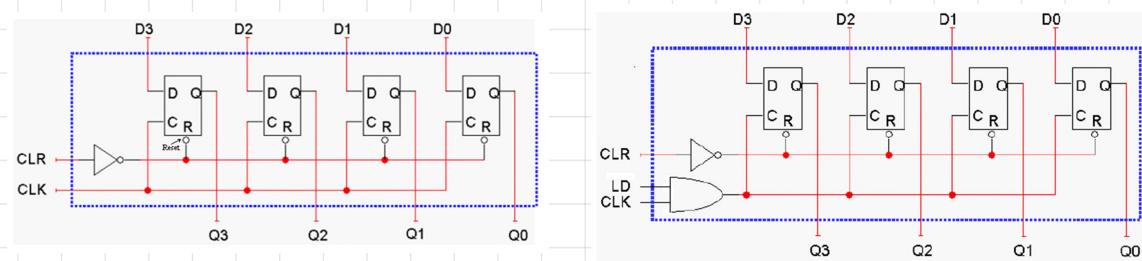
TP4: Registros y Contadores

REGISTROS

Entrada	Salida	Aplicación
Serie	Serie	Almacenamiento
Serie	Paralelo	Conversión
Paralelo	Serie	Conversión
Paralelo	Paralelo	Almacenamiento

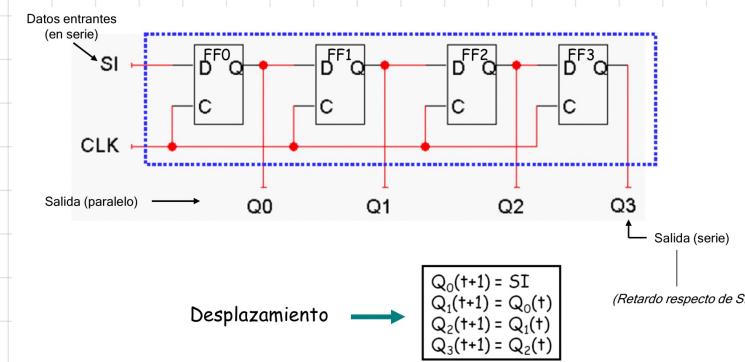
ALMACENA MIENTO

- ## • ff-D : PIPD

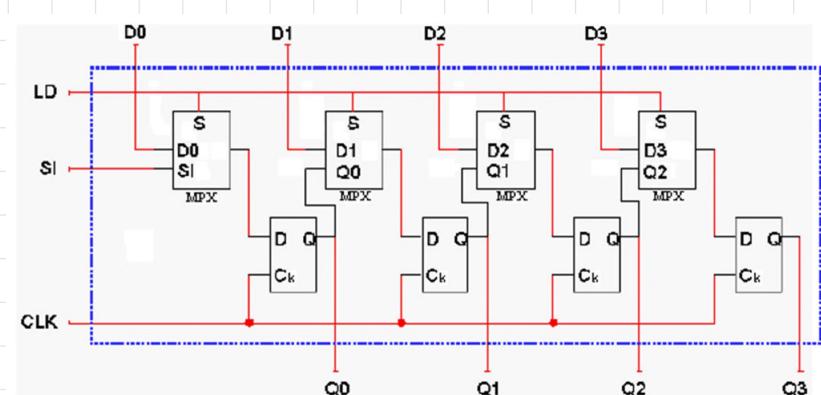


DESPLAZAMIENTO

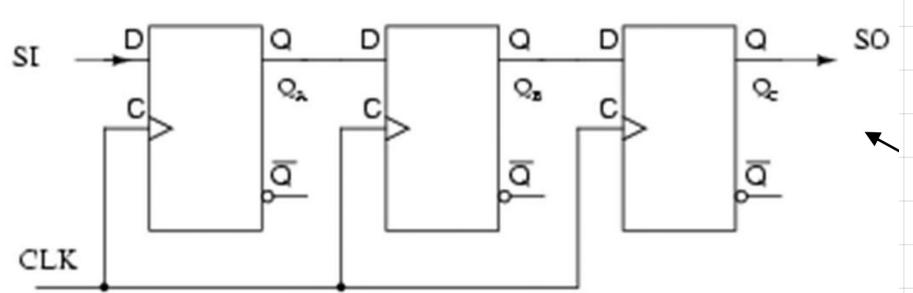
- FF-D : S1PO Corcado



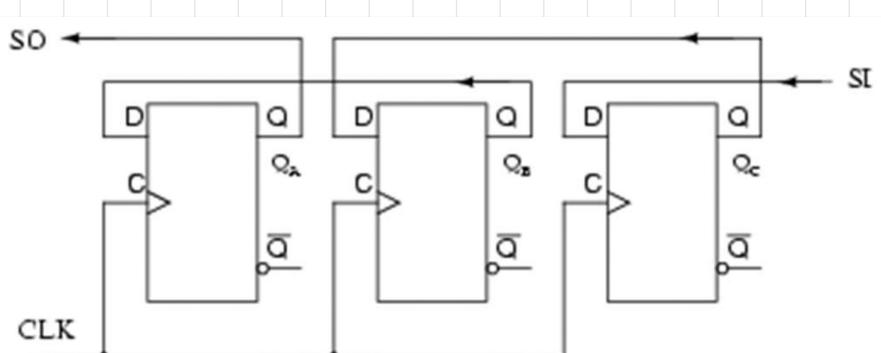
- Carga de datos en Paralelo



- FF- Δ : S1PO can cause (SHIFT RIGHT)



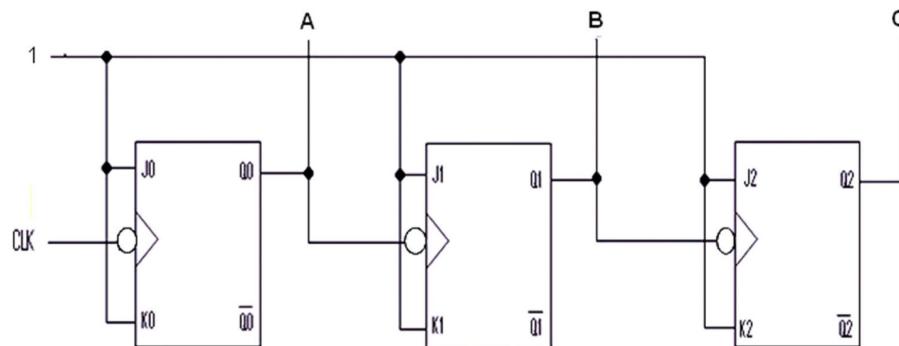
- FF - D . SI PO cascade invertida (SHIFT LEFT)



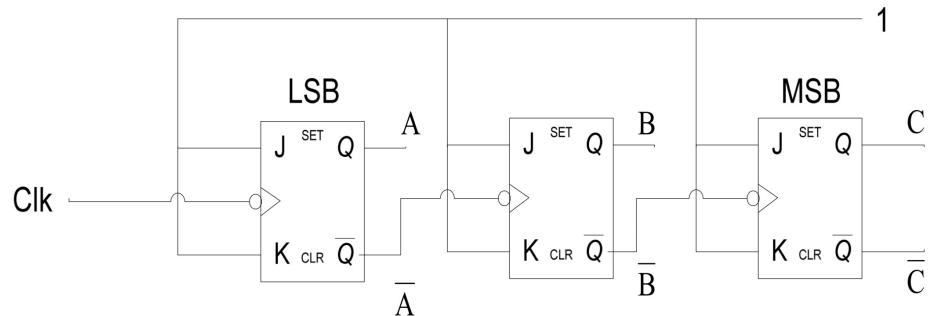
CONTADORES

ASÍNCRONOS

- FF-JK: $J=K=1$ SO Corriente

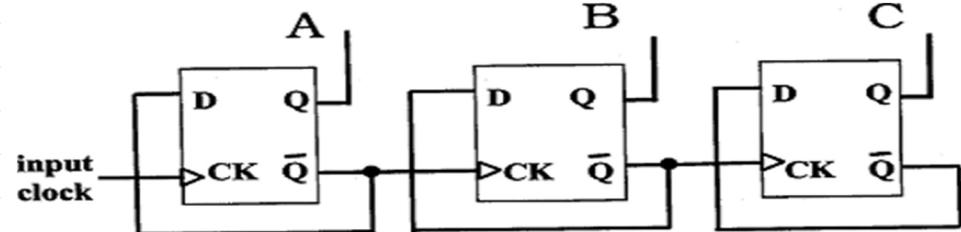


- FF-JK: $J=K=1$ SO Corriente Negada



Módulo: $2^3 = 8$

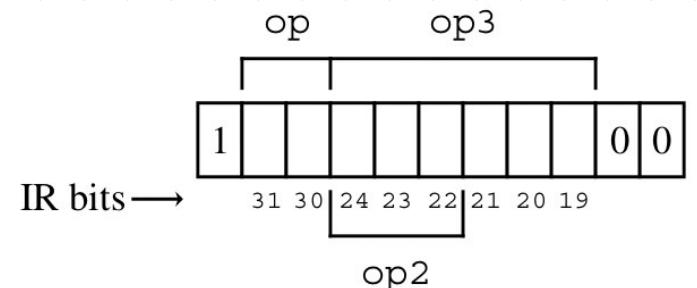
- FF-D: P1 Corriente Negada



SÍNCRONOS

- 1) Elijo módulo y código de cuenta (diagramma de estados)
- 2) Módulo \rightarrow Comtidad en flip flops mecanos.
- 3) Eligo un tipo de ff.
- 4) Obtener la lógica de compuertas en base al código de cuenta.

F_3	F_2	F_1	F_0	Operation	Changes	Condition	Codes
0	0	0	0	ANDCC (A, B)		yes	
0	0	0	1	ORCC (A, B)		yes	
0	0	1	0	NORCC (A, B)		yes	
0	0	1	1	ADDCC (A, B)		yes	
0	1	0	0	SRL (A, B)		no	
0	1	0	1	AND (A, B)		no	
0	1	1	0	OR (A, B)		no	
0	1	1	1	NOR (A, B)		no	
1	0	0	0	ADD (A, B)		no	
1	0	0	1	LSHIFT2 (A)		no	
1	0	1	0	LSHIFT10 (A)		no	
1	0	1	1	SIMM13 (A)		no	
1	1	0	0	SEXT13 (A)		no	
1	1	0	1	INC (A)		no	
1	1	1	0	INCPC (A)		no	
1	1	1	1	RSHIFT5 (A)		no	



```

0: R[ir] ← AND(R[pc], R[pc]); READ;
1: DECODE;
   / sethi
1152: R[rd] ← LSHIFT10(ir); GOTO 2047;
      / call
1280: R[15] ← AND(R[pc], R[pc]);
1281: R[temp0] ← ADD(R[ir], R[ir]);
1282: R[temp0] ← ADD(R[temp0], R[temp0]);
1283: R[pc] ← ADD(R[pc], R[temp0]);
      GOTO 0;
      / addcc
1600: IF R[IR[13]] THEN GOTO 1602;
1601: R[rd] ← ADDCC(R[rs1], R[rs2]);
      GOTO 2047;
1602: R[temp0] ← SEXT13(R[ir]);
1603: R[rd] ← ADDCC(R[rs1], R[temp0]);
      GOTO 2047;
      / andcc
1604: IF R[IR[13]] THEN GOTO 1606;
1605: R[rd] ← ANDCC(R[rs1], R[rs2]);
      GOTO 2047;
1606: R[temp0] ← SIMM13(R[ir]);
1607: R[rd] ← ANDCC(R[rs1], R[temp0]);
      GOTO 2047;
      / orcc
1608: IF R[IR[13]] THEN GOTO 1610;
1609: R[rd] ← ORCC(R[rs1], R[rs2]);
      GOTO 2047;
1610: R[temp0] ← SIMM13(R[ir]);
1611: R[rd] ← ORCC(R[rs1], R[temp0]);
      GOTO 2047;
      / orncc
1624: IF R[IR[13]] THEN GOTO 1626;
1625: R[rd] ← NORCC(R[rs1], R[rs2]);
      GOTO 2047;
1626: R[temp0] ← SIMM13(R[ir]);
1627: R[rd] ← NORCC(R[rs1], R[temp0]);
      GOTO 2047;
      / srl
1688: IF R[IR[13]] THEN GOTO 1690;
1689: R[rd] ← SRL(R[rs1], R[rs2]);
      GOTO 2047;
1690: R[temp0] ← SIMM13(R[ir]);
1691: R[rd] ← SRL(R[rs1], R[temp0]);
      GOTO 2047;
      / jmpl
1760: IF R[IR[13]] THEN GOTO 1762;
1761: R[pc] ← ADD(R[rs1], R[rs2]);
      GOTO 2

```

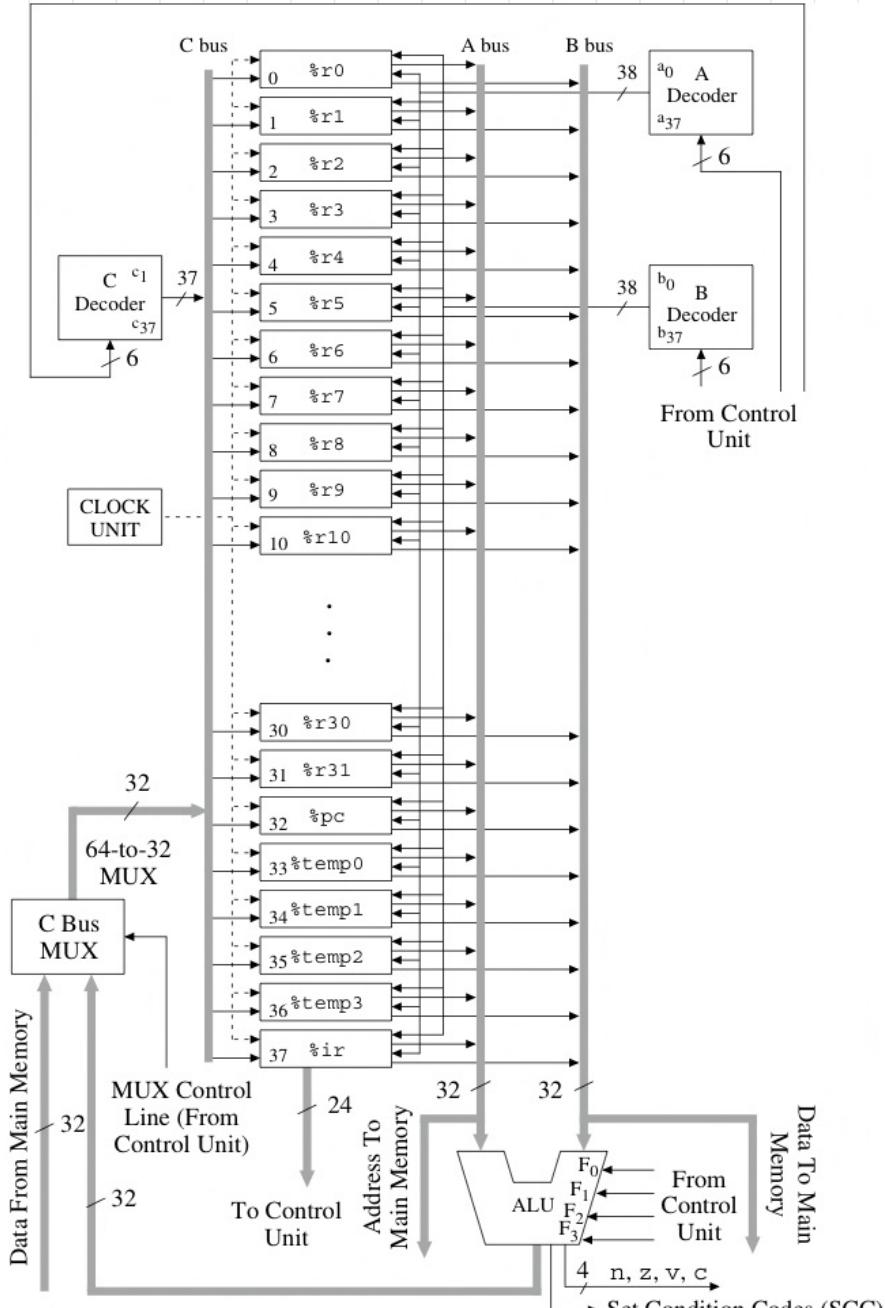
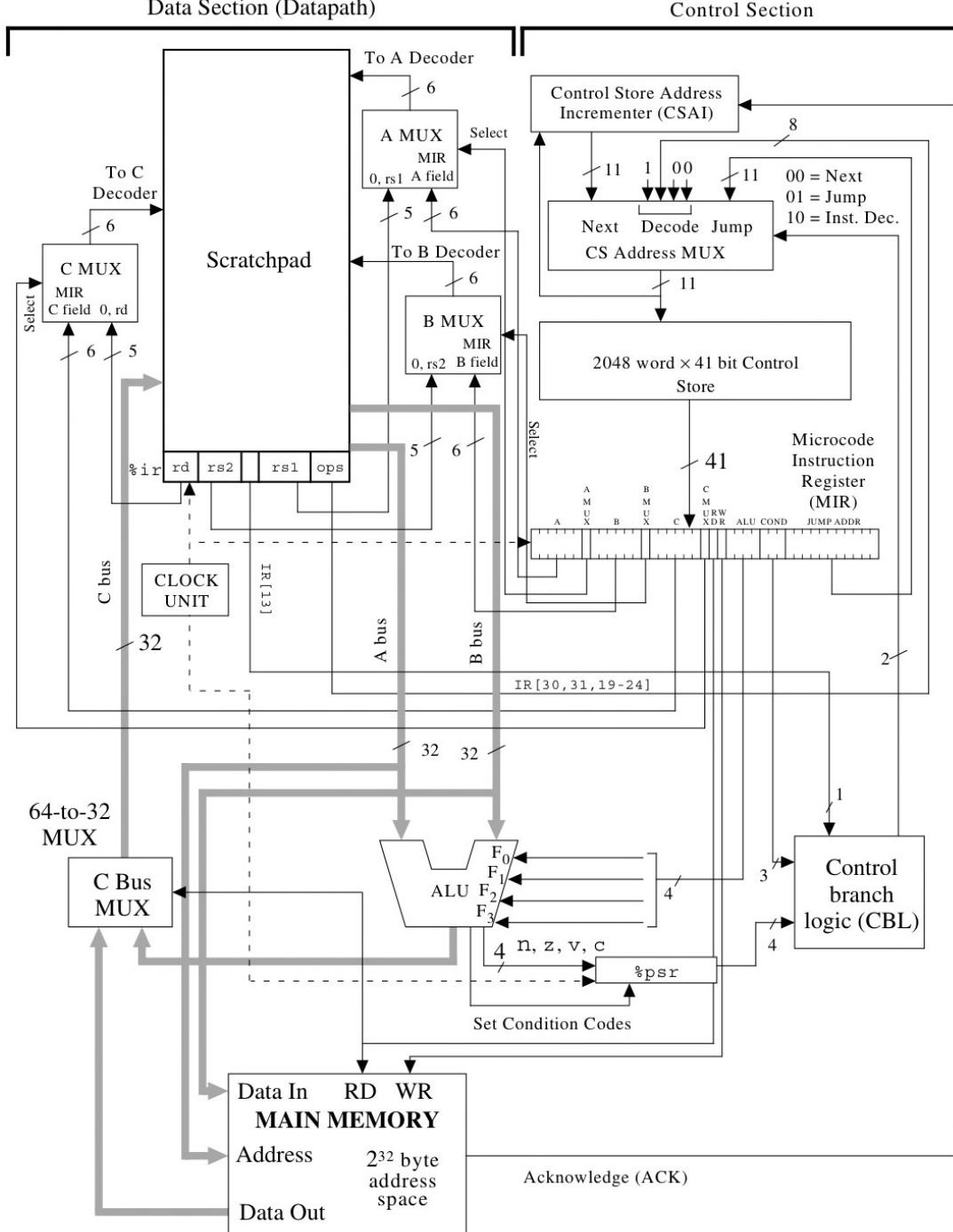
```

1762: R[temp0] ← SEXT13(R[ir]);
1763: R[pc] ← ADD(R[rs1], R[temp0]);
      GOTO 0;
      / ld
1792: R[temp0] ← ADD(R[rs1], R[rs2]);
      IF R[IR[13]] THEN GOTO 1794;
1793: R[rd] ← AND(R[temp0], R[temp0]);
      READ; GOTO 2047;
1794: R[temp0] ← SEXT13(R[ir]);
1795: R[temp0] ← ADD(R[rs1], R[temp0]);
      GOTO 1793;
      / st
1808: R[temp0] ← ADD(R[rs1], R[rs2]);
      IF R[IR[13]] THEN GOTO 1810;
1809: R[ir] ← RSHIFT5(R[ir]); GOTO 40;
  40: R[ir] ← RSHIFT5(R[ir]);
  41: R[ir] ← RSHIFT5(R[ir]);
  42: R[ir] ← RSHIFT5(R[ir]);
  43: R[ir] ← RSHIFT5(R[ir]);
  44: R[0] ← AND(R[temp0], R[rs2]);
      WRITE; GOTO 2047;
1810: R[temp0] ← SEXT13(R[ir]);
1811: R[temp0] ← ADD(R[rs1], R[temp0]);
      GOTO 1809;
      / Branch instructions: ba, be, bcs,
1088: GOTO 2;                                bvs, bneq
  2: R[temp0] ← LSHIFT10(R[ir]);
  3: R[temp0] ← RSHIFT5(R[temp0]);
  4: R[temp0] ← RSHIFT5(R[temp0]);
  5: R[ir] ← RSHIFT5(R[ir]);
  6: R[ir] ← RSHIFT5(R[ir]);
  7: R[ir] ← RSHIFT5(R[ir]);
  8: IF R[IR[13]] THEN GOTO 12;
      R[ir] ← ADD(R[ir], R[ir]);
  9: IF R[IR[13]] THEN GOTO 13;
      R[ir] ← ADD(R[ir], R[ir]);
 10: IF Z THEN GOTO 12;
      R[ir] ← ADD(R[ir], R[ir]);
 11: GOTO 2047;
 12: R[pc] ← ADD(R[pc], R[temp0]);
      GOTO 0;
 13: IF R[IR[13]] THEN GOTO 16;
      R[ir] ← ADD(R[ir], R[ir]);
 14: IF C THEN GOTO 12;
 15: GOTO 2047;
 16: IF R[IR[13]] THEN GOTO 19;
 17: IF N THEN GOTO 12;
 18: GOTO 2047;
 19: IF V THEN GOTO 12;
 20: GOTO 2047;
2047: R[pc] ← INCPC(R[pc]): GOTO 0;

```

C_2 C_1 C_0	Operation
0 0 0	Use NEXT ADDR
0 0 1	Use JUMP ADDR if $n = 1$
0 1 0	Use JUMP ADDR if $z = 1$
0 1 1	Use JUMP ADDR if $v = 1$
1 0 0	Use JUMP ADDR if $c = 1$
1 0 1	Use JUMP ADDR if $IR[13] = 1$
1 1 0	Use JUMP ADDR
1 1 1	DECODE

Mnemonic	Meaning
ld	Load a register from memory
st	Store a register into memory
sethi	Load the 22 most significant bits of a register
andcc	Bitwise logical AND
orcc	Bitwise logical OR
orncc	Bitwise logical NOR
srl	Shift right (logical)
addcc	Add
call	Call subroutine
jmpl	Jump and link (return from subroutine call)
be	Branch if equal
bneg	Branch if negative
bcs	Branch on carry
bvs	Branch on overflow
beq	Branch always



ARITHMETIC-LOGIC		CONTROL		DISPLACEMENT		MEMORY	
Instruction	Syntax	Instruction	Syntax	Instruction	Syntax	Instruction	Syntax
<u>add</u>	add reg_rs1, reg_or_imm, reg_rd	<u>ba</u>	ba label	<u>sll</u>	sll reg_rs1, reg_or_imm, reg_rd	<u>st</u>	st reg_rd, [address]
<u>addcc</u>	addcc reg_rs1, reg_or_imm, reg_rd	<u>bn</u>	bn label	<u>srl</u>	srl reg_rs1, reg_or_imm, reg_rd	<u>stb</u>	stb reg_rd, [address]
<u>sub</u>	sub reg_rs1, reg_or_imm, reg_rd	<u>bne</u>	bne label	<u>sra</u>	sra reg_rs1, reg_or_imm, reg_rd	<u>sth</u>	sth reg_rd, [address]
<u>subcc</u>	subcc reg_rs1, reg_or_imm, reg_rd	<u>be</u>	be label			<u>ld</u>	ld [address], reg_rd
<u>and</u>	and reg_rs1, reg_or_imm, reg_rd	<u>bg</u>	bg label			<u>ldsb</u>	ldsb [address], reg_rd
<u>andcc</u>	andcc reg_rs1, reg_or_imm, reg_rd	<u>ble</u>	ble label			<u>ldsh</u>	ldsh [address], reg_rd
<u>andn</u>	andn reg_rs1, reg_or_imm, reg_rd	<u>bge</u>	bge label			<u>ldub</u>	ldub [address], reg_rd
<u>andncc</u>	andncc reg_rs1, reg_or_imm, reg_rd	<u>bl</u>	bl label			<u>lduh</u>	lduh [address], reg_rd
<u>or</u>	or reg_rs1, reg_or_imm, reg_rd	<u>bgu</u>	bgu label				
<u>orcc</u>	orcc reg_rs1, reg_or_imm, reg_rd	<u>bleu</u>	bleu label				
<u>orn</u>	orn reg_rs1, reg_or_imm, reg_rd	<u>bcc</u>	bcc label				
<u>orncc</u>	orncc reg_rs1, reg_or_imm, reg_rd	<u>bcs</u>	bcs label				
<u>xor</u>	xor reg_rs1, reg_or_imm, reg_rd	<u>bpos</u>	bpos label				
<u>xorcc</u>	xorcc reg_rs1, reg_or_imm, reg_rd	<u>bneg</u>	bneg label				
<u>xnor</u>	xnor reg_rs1, reg_or_imm, reg_rd	<u>bvc</u>	bvc label				
<u>xnorcc</u>	xnorcc reg_rs1, reg_or_imm, reg_rd	<u>bvs</u>	bvs label				
<u>sethi</u>	sethi const22, reg_rd	<u>call</u>	call label				
		<u>impl</u>	impl address, reg_rd				

sll							
1 0	rd	1 0 0 1 0 1	rs1	0 0 0 0 0 0 0 0 0	rs2		
1 0	rd	1 0 0 1 0 1	rs1	1 0 0 0 0 0 0 0 0	shcnt		

ld							
1	1	rd	0 0 0 0 0 0	rs1	0	0 0 0 0 0 0 0 0	rs2
1	1	rd	0 0 0 0 0 0	rs1	1	simm13	

srl							
1 0	rd	1 0 0 1 1 0	rs1	0 0 0 0 0 0 0 0 0 0	rs2		
1 0	rd	1 0 0 1 1 0	rs1	1 0 0 0 0 0 0 0 0 0	shcnt		

ldsbt							
1	1	rd	0 0 1 0 0 1	rs1	0	0 0 0 0 0 0 0 0	rs2
1	1	rd	0 0 1 0 0 1	rs1	1		simm13

sra						
1 0	rd	1 0 0 1 1 1	rs1	0 0 0 0 0 0 0 0 0	rs2	
1 0	rd	1 0 0 1 1 1	rs1	1 0 0 0 0 0 0 0 0	shcnt	

ldsh						
1 1	rd	0 0 1 0 1 0	rs1	0	0 0 0 0 0 0 0	rs2
1 1	rd	0 0 1 0 1 0	rs1	1		simm13

st						
1 1	rd	0 0 0 1 0 0	rs1	0	0 0 0 0 0 0 0 0 0	rs2
1 1	rd	0 0 0 1 0 0	rs1	1	simm13	

ldub						
1 1	rd	0 0 0 0 0 1	rs1	0	0 0 0 0 0 0 0 0	rs2
1 1	rd	0 0 0 0 0 1	rs1	1	simm13	

sth						
1 1	rd	0 0 0 1 0 1	rs1	0 0 0 0 0 0 0 0 0	rs2	
1 1	rd	0 0 0 1 0 1	rs1	1		simm13

lduh						
1	1	rd	0 0 0 0 1 0	rs1	0	0 0 0 0 0 0 0 0
1	1	rd	0 0 0 0 1 0	rs1	1	simm13

CHAPTER 4: MACHINE LANGUAGE AND ASSEMBLY LANGUAGE

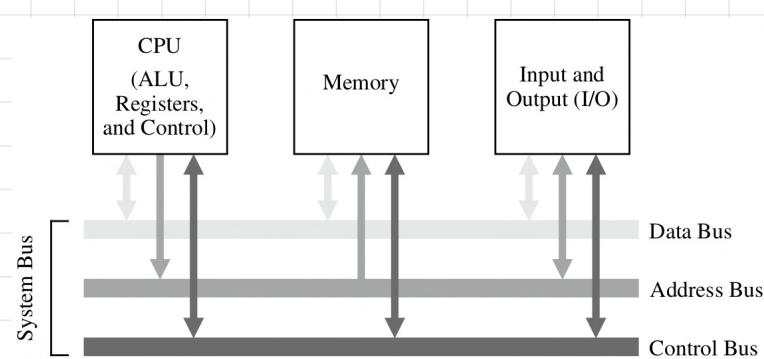
- ISA (Instruction Set Architecture): The ISA view corresponds to the Assembly language / Machine Code level.

It is between the high level language (none of the machine hardware is visible or of concern) and the control level (machine instructions are interpreted as register transfer actions) at the functional Unit level.

HARDWARE COMPONENTS OF THE ISA

The CPU interacts with its internal main memory and perform input and output with the outside world

THE SYSTEM BUS MODEL:



The CPU is interconnected with its memory and I/O systems

via shared system bus.

The CPU generates addresses that are placed onto the address bus, and the memory receives addresses from the address bus.

User writes a high level program → 4 compiler translates it into assembly language → An assembler translates the assembly language → into machine code → The machine code is stored on disk → The machine code is loaded from disk → The program is executed

During program execution, each instruction is brought into the ALU from the memory (one instruction at a time). The output of the program is placed, by a control unit, on a device such as video display or a disk. Communication among the three components (CPU, Memory, I/O) is handled with buses.

MEMORY:

Computer memory consists of a collection of consecutively numbered (addressed) registers, each one of which normally holds one byte

Each register has an address, referred to as memory location.

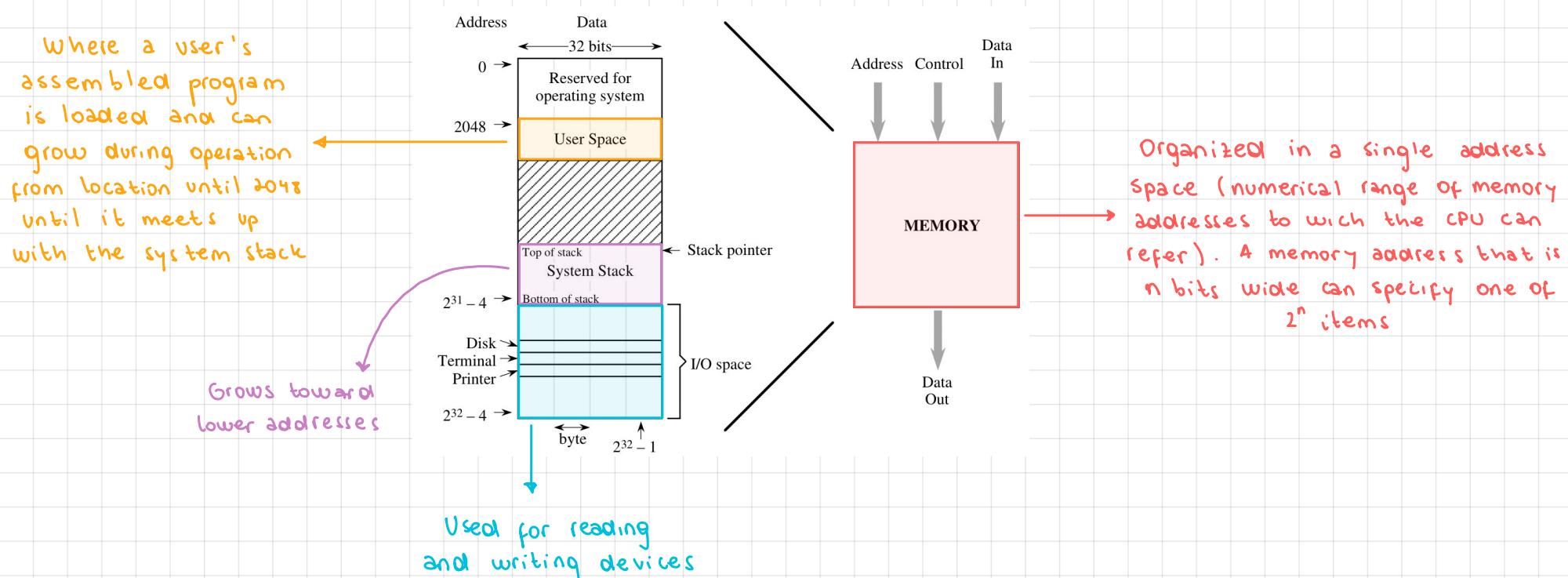
The meaning of word depends upon the particular processor (typical are 16, 32, 64 and 128 bits)

In a byte-addressable machine, the smallest object that can be referenced in memory is the byte, although there are usually instructions that read and write multi-byte words.

When multi-byte words are used, there are two types of orders in which the bytes are stored:

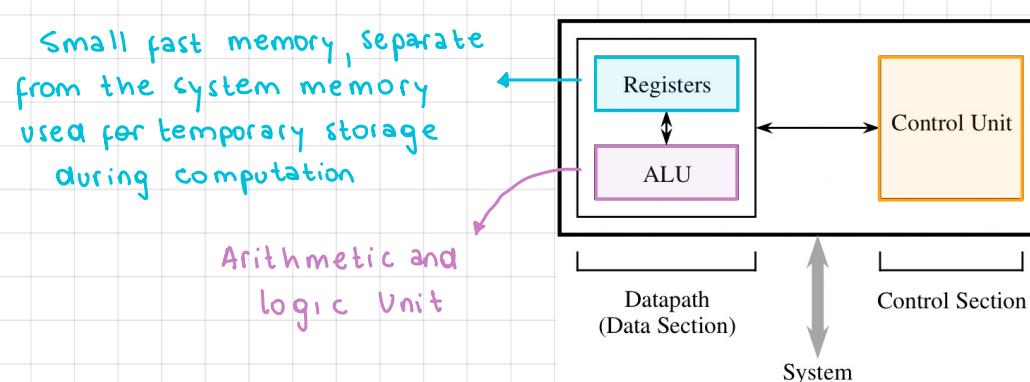
- BIG-ENDIAN: Most significant byte at lowest address.
- LITTLE-ENDIAN: least significant byte at lowest address.

The highest address is one less than the size of the memory.



THE CPU :

Consists of a **data section** (that contains registers and an ALU) and a **control section** (which interprets instructions and effects register transfers)



Executes the program instructions which are stored in the main memory by the fetch-execute cycle:

- ① Fetch the next instruction to be executed from memory.
- ② Decode the opcode
- ③ Read operand(s) from main memory, if any.
- ④ Execute the instructions and store results.
- ⑤ Go to step ①.

There are two registers that form the interface between the control unit and the data unit

known as the **program counter** (PC) and the **instruction register** (IR).

Contains the address of the instruction being executed

the instruction that is pointed to by the PC is fetched from the memory and is stored in the IR where it's interpreted

The ALU implements a variety of binary (two-operand) and unary (one-operand) operations. Operations and operands to be used during the operations are selected by the Control Unit. The two source operands are fetched from the register file onto buses labeled "Register Source 1 (rs1)" and "Register Source 2 (rs2)". The output from the ALU is placed on the bus labeled "Register Destination (rd)" where the results are conveyed back to the register file.

- **THE INSTRUCTION SET**: The collection of instructions that a processor can execute. It defines it
- **COMPILER**: Computer program that transforms programs written in a high-level language into machine language.

In the process of **compiling** a program (translation process), a high-level source program is transformed into **assembly language**, and the assembly language is then translated into machine code for the target machine by an assembler. It takes place at **compile time** and **assembly time** respectively.

The resulting object program can be linked with other object programs at **link time**. The linked program (usually stored on a disk) is loaded into main memory at **load time** and executed by the CPU at **run time**.

ARC: A RISC COMPUTER

- **SPARC (Scalable Processor Architecture)**: Processor that was developed at Sun Microsystems in the mid-1980's.
- **ARC (A RISC Computer)**: Has most of the important features of the SPARC architecture, but without some of the more complex features that are present in a commercial processor.

ARC MEMORY

The ARC is a 32-bit machine with byte-addressable memory.

Each integer is stored in memory as a collection of four bytes. ARC is a big-endian architecture. The largest possible byte address in the ARC is $2^{32}-1$, so the address of the highest word in the memory map is three bytes lower than this, or $2^{32}-4$.

ARC INSTRUCTION SET

- The ARC has 32 32-bit general-purpose registers, as well as PC and an IR.
- There is a **Processor Status Register** (PSR) that contains information about the state of the processor, including information about the results of arithmetic operations. The "arithmetic flags" in the PSR are called the **condition codes**. They specify whether a specified arithmetic operation resulted in a zero value (=), a negative value (n), a carry out from the 32-bit ALU (c), and an overflow (v) which is set when the results of the arithmetic operation are too large to be handled by the ALU.
- All instructions are one word (32 bits) in size.
- The ARC is a **load-store machine**: the only allowable memory access operations load a value into one of the registers, or store a value contained in one of the registers into a memory location. All arithmetic operations operate on values that are contained in registers, and the results are placed in a register. Each instruction is represented by a **mnemonic**, which is a name that represents the instruction.

ARC INSTRUCTION FORMATS

The **instruction format** defines how the various bit fields of an instruction are laid out by the assembler, and how they are interpreted by the ARC control unit. The five formats are:

op		31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00																										
SETHI Format	0 0	rd	op2	imm22																								
Branch Format	0 0 0	cond	op2	disp22																								
CALL format		0 1	disp30												i													
Arithmetic Formats		1 0	rd	op3	rs1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	rs2	31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00																				
Memory Formats		1 0	rd	op3	rs1	1	simm13																					
op		1 1	rd	op3	rs1	1	simm13																					
PSR		1 1	rd	op3	rs1	1	simm13																					
		31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00													n z v c													

- The **imm22** and **disp22** fields each hold a 22-bit constant that is used as the operand for the **SETHI** format (for **imm22**) or for calculating a displacement for a branch address (for a **disp22**).
- The **disp30** field contains a 30-bit displacement that is used in calculating the address of the called routine.
- The **simm13** field is a 13-bit immediate value that is sign extended to 32 bits for the second source when the **i** (immediate) field is 1.

ARC DATA FORMATS

The ARC supports 12 different data formats grouped into:

signed integer, unsigned integer and floating point.

Allowable formats widths are: **byte (8b)**, **half-word (16b)**,

word **single-word (32b)**, **tagged**

word (32b: 30 most significant from the value and 2 least from a tag),

doubleword (64b) and **quadword (128b)**.

Unsigned and signed integers

Floating Point Formats		
Floating Point Single	exponent	fraction
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00	23 22	0
Floating Point Double	exponent	fraction
63 62 52 51	32	
31	0	
Floating Point Quad	exponent	fraction
127 126	112 113	96
95	64	
63	32	
31	0	

Signed Formats	
Signed Integer Byte	[S] 7 6 0
Signed Integer Halfword	[S] 15 14 0
Signed Integer Word	[S] 31 30 0
Signed Integer Double	[S] 63 62 32
	31 0

Unsigned Formats	
Unsigned Integer Byte	[] 7 0
Unsigned Integer Halfword	[] 15 0
Unsigned Integer Word	[] 31 0
Tagged Word	[] 31 2 1 0
Unsigned Integer Double	[] 63 32
	31 0

are both stored and manipulated as 2's complement integers but it's their interpretation that varies. One subset of the branch instructions assumes that the value(s) being compared are signed integers while the other assumes they are unsigned, so then the c bit indicates unsigned integer overflow and the v bit signed overflow.

The tagged word uses the 2 least significant bits to indicate overflow in which an attempt is made to store a value that is larger than 30 bits into the allocated 30b of the 32b word.

ARC INSTRUCTION DESCRIPTION.

- ld : ld [x], %r1

Copy the contents of memory location x into register %r1.

- st : st %r1, [x]

Copy the contents of register %r1 into memory location x.

- sethi : sethi 0x304F15, %r1

Set the high 22 bits of %r1 to $304F15_{16}$ and set the low 10 bits to zero.

- andcc : andcc %r1, %r2, %r3

Logically AND %r1 and %r2 and place the result in %r3

- orcc : orcc %r1, 1, %r1

Set the least significant bit of %r1 to 1.

- orncc : orncc %r1, %r0, %r1

Complement %r1.

- srl : srl %r1, 3, %r2

Shift %r1 right by three bits and store it in %r2. Zeros are copied into the three most significant bits of %r2.

- addcc : addcc %r1, 5, %r1

Add 5 to %r1.

- call : call sub-r

Call a subroutine that begins at location sub-r.

- jmp : jmp %r15+4, %r0

Return from subroutine. The value of the PC for the call instruction was previously saved in %r15 and so the return address should be computed for the instruction that follows the call, at %r15+4. The current address is discarded in %r0.

- be be label

Branch to label if $z=1$.

- bneg : bneg label

Branch to label if $n=1$.

- bcs : bcs label

Branch to label if $c=1$.

• bvs : bvs label

~~

Branch to label if V=1.

• ba : ba label

~~

Branch to label regardless of the settings of the condition codes.

PSEUDO-OPS

Pseudo-Op	Usage	Meaning
.equ	X .equ #10	Treat symbol X as (10) ₁₆
.begin	.begin	Start assembling
.end	.end	Stop assembling
.org	.org 2048	Change location counter to 2048
.dwb	.dwb 25	Reserve a block of 25 words
.global	.global Y	Y is used in another module
.extern	.extern Z	Z is defined in another module
.macro	.macro M a, b, ...	Define macro M with formal parameters a, b, ...
.endmacro	.endmacro	End of macro definition
.if	.if <cond>	Assemble if <cond> is true
.endif	.endif	End of .if construct

Not opcodes but instructions to the assembler to perform some action at assembly time

Unlike the processor opcodes (specific to a given machine), pseudo-ops are specific to a given assembler because they are executed by the assembler itself.

SUBROUTINE LINKAGE AND STACKS

A **subroutine** (AKA function or procedure) is a sequence of instructions invoked in a manner that makes it appear to be a single instruction in a high level view. When a program calls it, control is passed from the program to the subroutine which then returns to the location just past where it was called.

The methods for passing arguments to and from the called routine are called **calling conventions** and the process of passing arguments between routines is referred to as **subroutine linkage**.

CALLING CONVENTIONS:

① **Using Registers:** Places the arguments in registers.

② **Using Data link area:** The address of the data link area is passed in a register to the called routine.

③ **Using a Stack:** The calling routine pushes all of its arguments onto a LIFO stack. The called routine then pops the passed arguments from the stack, and pushes any return values onto the stack. The calling routine then retrieves the return value(s) from the stack and continues execution.

CHAPTER 5: LANGUAGES AND THE MACHINE

THE COMPILATION PROCESS

THE STEPS OF COMPILEATION:

- **Lexical Analysis:** Reduce the program text to the basic symbols of the language.
- **Syntactic Analysis:** Parse the symbols to recognize the underlying program structure.
- **Name Analysis:** Associate the names with program variables and then with memory locations.
- **Type Analysis:** Determine the type of all data items.
- **Action Mapping and Code Generation:** Associate program statements with their assembly language sequence.
- **Additional Steps:** Allocate variables to registers, track register usage, optimize the program.

THE COMPILER MAPPING SPECIFICATION:

It's the information about the particular ISA for the compiler.

HOW THE COMPILER MAPS THE THREE INSTRUCTIONS CLASSES INTO ASSEMBLY CODE:

Only **global variables** (static variables) have addresses that are known at compile time.

local variables (automatic variables) only come to existence when the function or block is entered and they disappear when the function or block is exited for the last time.

DATA MOVEMENT:

① Structures

```
struct point {  
    int x;  
    int y;  
    int z;  
}
```

The compiler would lay out this structure in memory as three consecutive memory locations.

struct point pt; \Rightarrow at $pt+8$.

② Arrays

```
int A[10];
```

The array index can be computed at run time.

The general expression for computing the machine address of an array element at run time is given

by : Element Address = $BASE + (INDEX \times START) * SIZE$

Starting address of the array Index of the desired element Size of an individual element in bytes Starting index of the array

```
sub %13, %14, %16    ! %16  $\leftarrow$  INDEX-START  
sll %16, 2, %16      ! %16 * 4  
ld [%16], %11        ! array value
```

It costs more instructions if $SIZE \neq$ power of 2

ARITHMETIC INSTRUCTIONS:

When the compiler finds an arithmetic instruction that requires more registers than are available in the machine, it must temporarily store variables on the stack, called **register spill**.

To decide when the value in a register is no longer needed to be stored, compilers use a technique known as **live-dead analysis**.

THE ASSEMBLY PROCESS

Commercial assemblers provide at least the following capabilities:

- Allow the programmer to specify the run-time location of data values and programs.

- Provide a means for the programmer to initialize data values in memory prior to program execution.
- Provide assembly-language mnemonics for all machine instructions and addressing modes, and translate valid assembly language statements into their equivalent machine language binary values.
- Permit the use of symbolic labels to represent address and constants.
- Provide a means for the programmer to specify the starting address of the program, if there is one.
- Provide a degree of assemble-time arithmetic.
- Include a mechanism that allows variables to be defined in one assembly language program and used in another, separately assembled program.
- Provide for the expansion of **macro routines** (routines that can be defined once and then instantiated as many times as needed).

ASSEMBLY AND TWO PASS ASSEMBLERS:

The first pass is dedicated to determining the address of all data items and machine instructions, and selecting which machine instruction should be produced for each assembly language instructions.

The address of data items and instructions are determined by the **location counter** which keeps track of the address of the current instruction or data item as assembly proceeds. The **.org** pseudo-op sets the location counter to the value specified by current statement.

During this pass the assembler also inserts the definitions of all labels and constant values into a table, referred as the **symbol table**.

On the second pass the assembler allows symbols to be used in the program before they are defined, known as **forward referencing**, generating the machine code and inserting the values of symbols.

FINAL TASKS OF THE ASSEMBLER:

After assembly is complete the assembler must add additional information for the linker and loader:

- The module name and size.
- The address of the start symbol.
- The addresses of global symbols and information about which symbols remain undefined in the module because they are defined as global in another.
- Information about any library routines that are referenced by the module.
- The values of any constants that are to be loaded into memory.
- Relocation information.

LINKING AND LOADING

A linkage editor (**linker**) is a software program that combines separately assembled programs (**object modules**) into a single program (**load module**).

LINKING:

The linker must:

- Resolve address references that are external to modules as it links them.
- Relocate each module by combining them end-to-end as appropriate.
- Specify the starting symbol of the load module.

- If the memory model includes more than one memory segment, the linker must specify the identities and contents of the various segments.

In resolving address references the linker needs to distinguish local from global symbol names: the .global pseudo-op instructs the assembler to mark a symbol as being available to other object modules during linking phase; while the .extern identifies a label that is used in one module but is defined in another.

The assembler marks symbols that may have their address changed during linking as relocatable. Absolute numbers (marked by .equ or that appear in memory locations) are not relocatable. Memory locations that are positioned relative to a .org statement are generally relocatable.

LOADING:

The loader is a software program that places the load module into main memory and initializes the stack pointer (.sp) and the program counter (.pc) to their initial values.

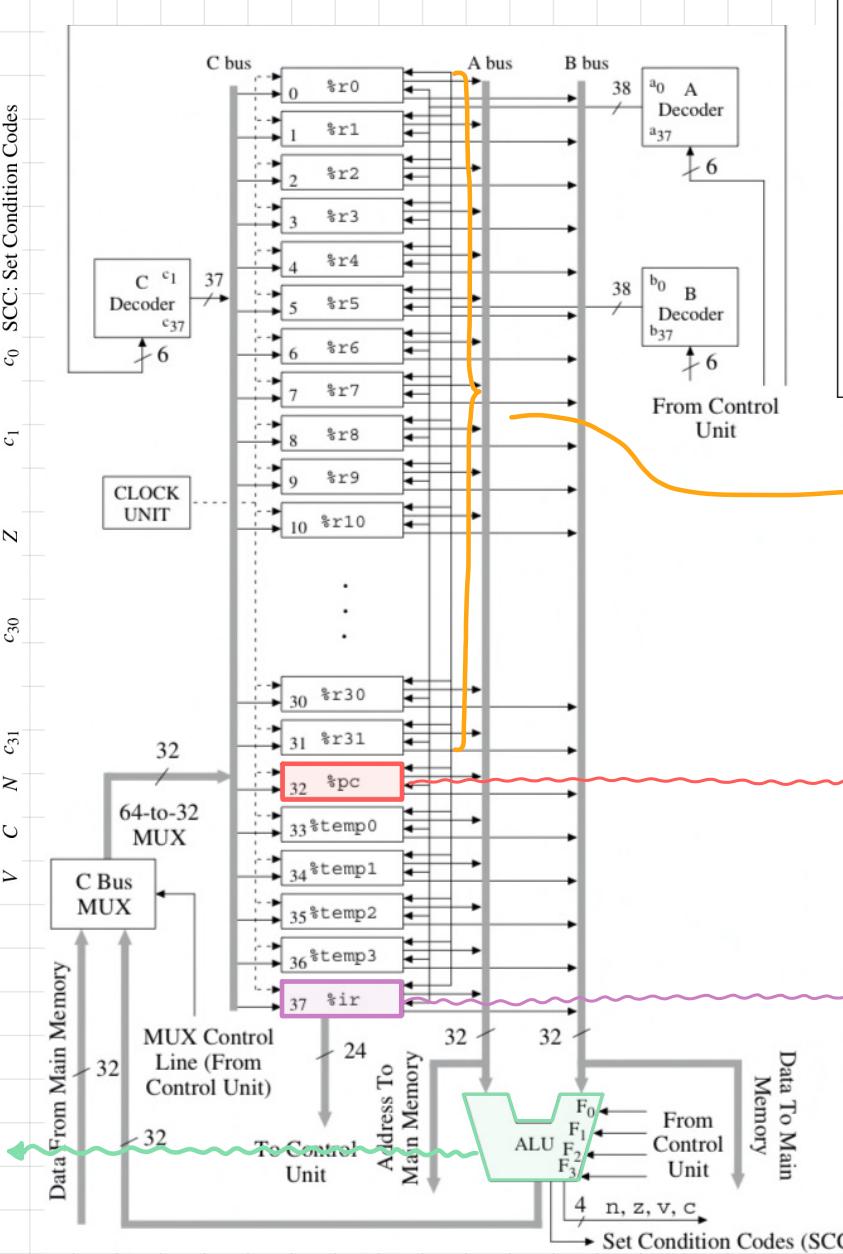
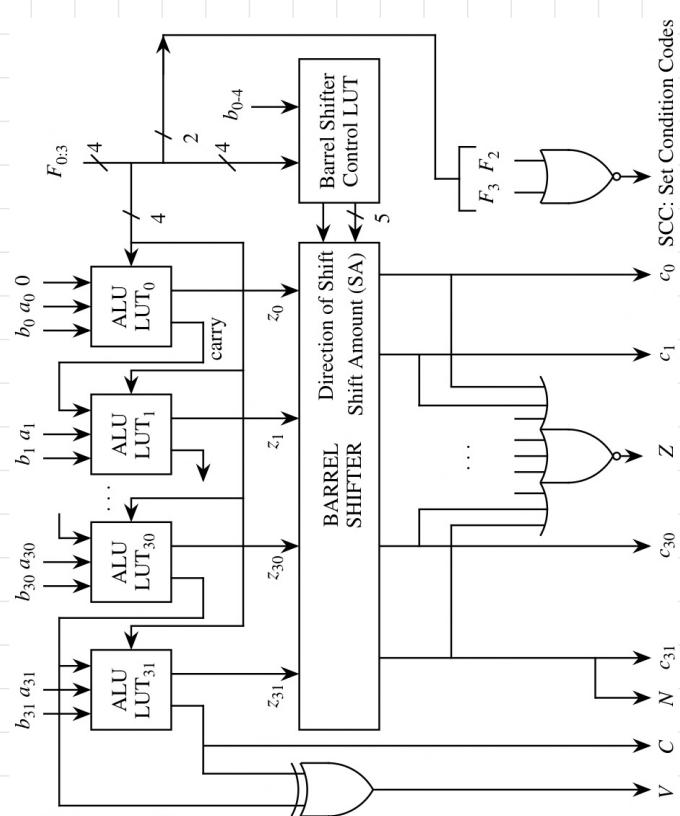
If there are several programs in memory at any time, the loader must relocate these modules at load time by adding an offset to all of the relocatable code in a module. This kind of loader is known as relocating loader.

It doesn't simply repeat the job of the linker, which has to combine several object modules into a single load module, but it modifies relocatable addresses within a single load module so that several programs can reside in memory simultaneously. A linking loader performs both the linking process and the loading process.

CHAPTER 6: DATAPATH AND CONTROL

A MICROARCHITECTURE FOR THE ARC

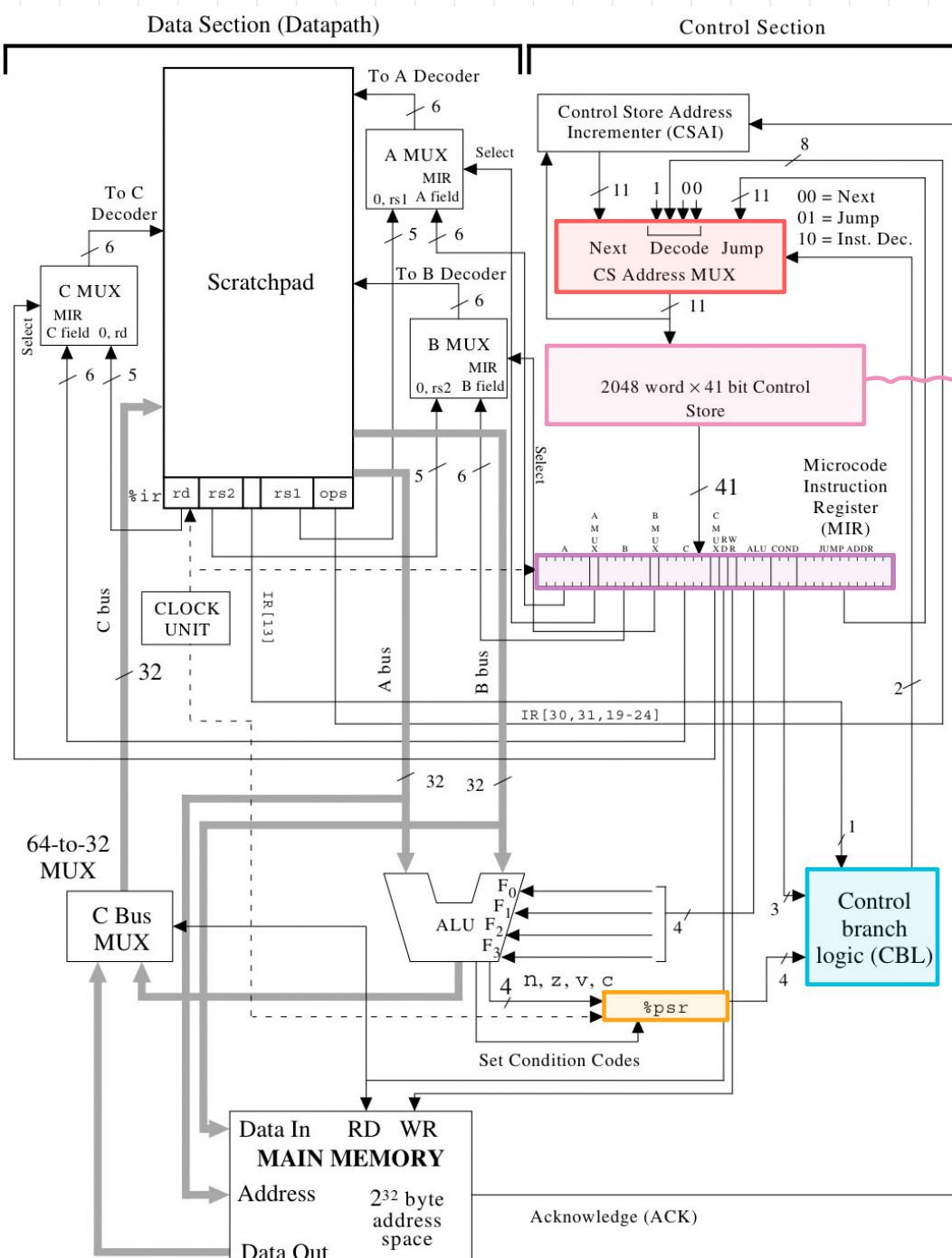
THE DATAPATH:



THE CONTROL SECTION:

When the micro-architecture begins operation, a reset circuit places the microword at location 0 in the control store into the MIR and executes it. From that point, a micro-

word is selected for execution from either the next, the decode or the jump inputs to the CS Address MUX according to



F_3	F_2	F_1	F_0	Operation	Changes Condition Codes
0	0	0	0	ANDCC (A, B)	yes
0	0	0	1	ORCC (A, B)	yes
0	0	1	0	NORCC (A, B)	yes
0	0	1	1	ADDCC (A, B)	yes
0	1	0	0	SRL (A, B)	no
0	1	0	1	AND (A, B)	no
0	1	1	0	OR (A, B)	no
0	1	1	1	NOR (A, B)	no
1	0	0	0	ADD (A, B)	no
1	0	0	1	LSHIFT2 (A)	no
1	0	1	0	LSHIFT10 (A)	no
1	0	1	1	SIMM13 (A)	no
1	1	0	0	SEXT13 (A)	no
1	1	0	1	INC (A)	no
1	1	1	0	INCPC (A)	no
1	1	1	1	RSHIFT5 (A)	no

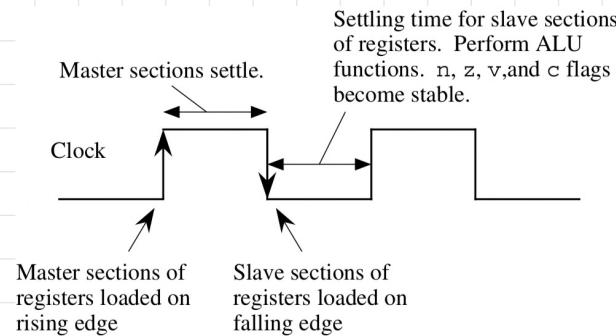
C_2	C_1	C_0	Operation
0	0	0	Use NEXT ADDR
0	0	1	Use JUMP ADDR if $n = 1$
0	1	0	Use JUMP ADDR if $z = 1$
0	1	1	Use JUMP ADDR if $v = 1$
1	0	0	Use JUMP ADDR if $c = 1$
1	0	1	Use JUMP ADDR if $\text{IR}[13] = 1$
1	1	0	Use JUMP ADDR if $\text{IR}[13] = 1$
1	1	1	DECODE

op op3

IR bits → 31 30 24 23 22 21 20 19 0 0

is placed in the MIR, the datapath performs operations according to the settings in the individual fields of the MIR.

TIMING:



While the clock is low, the ALU, CBL, and MUX functions are performed, which settle in time for the rising edge of the clock. On the rising edge of the clock, the new values of the registers are written into the masters sections. The registers settle while the clock is high.

DEVELOPING THE MICROPROGRAM.

```

0: R[ir] ← AND(R[pc], R[pc]); READ;
1: DECODE;
   / sethi
1152: R[rd] ← LSHIFT10(ir); GOTO 2047;
   / call
1280: R[15] ← AND(R[pc], R[pc]); 215=PC
1281: R[temp0] ← ADD(R[ir], R[ir]);
1282: R[temp0] ← ADD(R[temp0], R[temp0]);
1283: R[pc] ← ADD(R[pc], R[temp0]); PC+=4; ir
   GOTO 0;
   / addcc
1600: IF R[IR[13]] THEN GOTO 1602;
1601: R[rd] ← ADDCC(R[rs1], R[rs2]);
   GOTO 2047;
1602: R[temp0] ← SEXT13(R[ir]);
1603: R[rd] ← ADDCC(R[rs1], R[temp0]);
   GOTO 2047;
   / andcc
1604: IF R[IR[13]] THEN GOTO 1606;
1605: R[rd] ← ANDCC(R[rs1], R[rs2]);
   GOTO 2047;
1606: R[temp0] ← SIMM13(R[ir]);
1607: R[rd] ← ANDCC(R[rs1], R[temp0]);
   GOTO 2047;
   / orcc
1608: IF R[IR[13]] THEN GOTO 1610;
1609: R[rd] ← ORCC(R[rs1], R[rs2]);
   GOTO 2047;
1610: R[temp0] ← SIMM13(R[ir]);
1611: R[rd] ← ORCC(R[rs1], R[temp0]);
   GOTO 2047;
   / orncc
1624: IF R[IR[13]] THEN GOTO 1626;
1625: R[rd] ← NORCC(R[rs1], R[rs2]);
   GOTO 2047;
1626: R[temp0] ← SIMM13(R[ir]);
1627: R[rd] ← NORCC(R[rs1], R[temp0]);
   GOTO 2047;
   / srl
1688: IF R[IR[13]] THEN GOTO 1690;
1689: R[rd] ← SRL(R[rs1], R[rs2]);
   GOTO 2047;
1690: R[temp0] ← SIMM13(R[ir]);
1691: R[rd] ← SRL(R[rs1], R[temp0]);
   GOTO 2047;
   / jmp1
1760: IF R[IR[13]] THEN GOTO 1762;
1761: R[pc] ← ADD(R[rs1], R[rs2]);
   GOTO 0;

```

```

1762: R[temp0] ← SEXT13(R[ir]);
1763: R[pc] ← ADD(R[rs1], R[temp0]);
   GOTO 0;
   / ld
1792: R[temp0] ← ADD(R[rs1], R[rs2]);
   IF R[IR[13]] THEN GOTO 1794;
1793: R[rd] ← AND(R[temp0], R[temp0]);
   READ; GOTO 2047;
1794: R[temp0] ← SEXT13(R[ir]);
1795: R[temp0] ← ADD(R[rs1], R[temp0]);
   GOTO 1793;
   / st
1808: R[temp0] ← ADD(R[rs1], R[rs2]);
   IF R[IR[13]] THEN GOTO 1810;
1809: R[ir] ← RSHIFT5(R[ir]); GOTO 40;
40: R[ir] ← RSHIFT5(R[ir]);
41: R[ir] ← RSHIFT5(R[ir]);
42: R[ir] ← RSHIFT5(R[ir]);
43: R[ir] ← RSHIFT5(R[ir]);
44: R[0] ← AND(R[temp0], R[rs2]);
   WRITE; GOTO 2047;
1810: R[temp0] ← SEXT13(R[ir]);
1811: R[temp0] ← ADD(R[rs1], R[temp0]);
   GOTO 1809;
   / Branch instructions: ba, be, bcs,
   bvs, bneg
1088: GOTO 2;
2: R[temp0] ← LSHIFT10(R[ir]);
3: R[temp0] ← RSHIFT5(R[temp0]);
4: R[temp0] ← RSHIFT5(R[temp0]);
5: R[ir] ← RSHIFT5(R[ir]);
6: R[ir] ← RSHIFT5(R[ir]);
7: R[ir] ← RSHIFT5(R[ir]);
8: IF R[IR[13]] THEN GOTO 12;
   R[ir] ← ADD(R[ir], R[ir]);
9: IF R[IR[13]] THEN GOTO 13;
   R[ir] ← ADD(R[ir], R[ir]);
10: IF Z THEN GOTO 12;
   R[ir] ← ADD(R[ir], R[ir]);
11: GOTO 2047;
12: R[pc] ← ADD(R[pc], R[temp0]);
   GOTO 0;
13: IF R[IR[13]] THEN GOTO 16;
   R[ir] ← ADD(R[ir], R[ir]);
14: IF C THEN GOTO 12;
15: GOTO 2047;
16: IF R[IR[13]] THEN GOTO 19;
17: IF N THEN GOTO 12;
18: GOTO 2047;
19: IF V THEN GOTO 12;
20: GOTO 2047;
2047: R[pc] ← INCPC(R[pc]); GOTO 0;

```

determined by the cond field of the branch format (bits 25-28), which is not used during a DECODE operation.

In a microprogrammed architecture, instructions are interpreted by the microprogram (often referred to as firmware) because it bridges the gap between the hardware and the software) in the control store.

In the control store, each microstatement is stored in coded form in a single microword.

The first task in the execution of a user-level program is to bring the instruction pointed to by the PC from main memory into the IR so then the next step is to decode the opcode fields.

Additional decoding is needed for the branch instructions because the type of branch is

CHAPTER 7: MEMORY

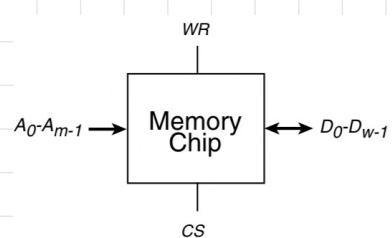
RANDOM ACCESS MEMORY (RAM)

Any memory location can be accessed in the same amount of time, regardless of its position in the memory.

- SRAM (Static RAM): RAM chips that are based upon flip-flops. The contents of each location persist as long as power is applied to the chips.
- DRAM (Dynamic RAM): RAM chips, employ a capacitor which stores a minute amount of electric charge, in which the charge level represents a 1 or a 0. Capacitors are much smaller than flip-flops, and so a DRAM can hold much more information in the same area than an SRAM. Since the charges on the capacitors dissipate with time, the charge in the capacitor storage cells in DRAMs must be restored or refreshed frequently.

Although DRAMs are very economical, SRAMs offer greater speed. The refresh cycles, error detection circuitry, and the low operation powers of DRAMs create a speed difference that is 1/4 of SRAM speed, but SRAMs also incur a significant cost.

CHIP ORGANIZATION



The address lines $A_0 - A_{m-1}$ in the RAM chip contain an address, which is decoded from an m -bit address into one of 2^m locations within the chip, each of which has a w -bit word associated with it. The chip contains $2^m \times w$ b.

READ ONLY MEMORY (ROM)

for some applications, the program never changes, and so it is hardwired into a ROM.

- PROM (Programmable ROM): Allow their contents to be written by a user with a device called PROM burner. It allows the designer to delay decisions about what information is stored. It can only be written once, or can be rewritten only if the existing pattern is a subset of the new pattern.
- EPROM (Erasable ROM): Can be written several times, after being erased with ultraviolet light (UVPROMs).
- EEPROM (Electrically Erasable ROM): Allow their contents to be rewritten electrically.

PROMs will be used for control units and for arithmetic logic units (ALUs), generating a truth table with all possible combinations of operands and all combinations of functions, and send the truth table to a PROM burner which loads it into the PROM. This approach is called look up table (LUT).

CACHE MEMORY

- LOCALITY PRINCIPLE: When a program executes on a computer, most of the memory references are made to a small number of locations.
- TEMPORAL LOCALITY: When a program references to a memory location, it is likely to reference that same memory location again soon.
- SPATIAL LOCALITY: A memory location that is near a recently referenced location is more likely to be referenced than a memory location that is farther away.

A small but fast cache memory (in which the contents of the most commonly accessed locations are

mantained) can be placed between the main memory and the CPU. When a program executes, the cache memory is searched first, and the referenced word is accessed in the cache if the word is present. If the referenced word is not in the cache, then a free location is created in the cache and the referenced word is brought into the cache from the main memory. The word is then accessed in the cache.

The cache is placed both physically and logically closer to the CPU than the main memory, avoiding communication delays over a shared bus.

MAPPED CACHE:

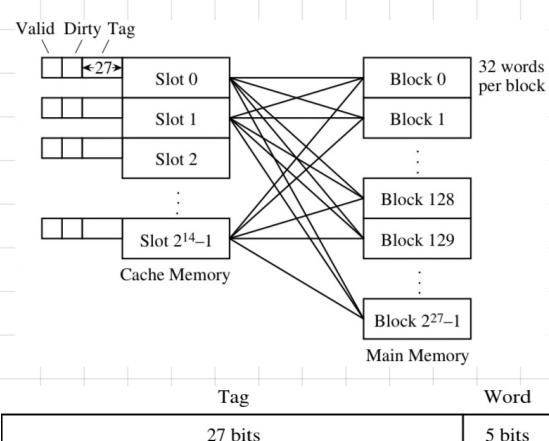
To keep track of which one of the possible blocks is in each slot, a **tag** field is added to each slot.

All the tags are stored in a special tag memory where they can be searched in parallel.

When a program is first loaded into main memory, the cache is cleared, and so while a program is executing, a **valid** bit is needed to indicate whether or not the slot holds a block that belongs to the program being executed. The **dirty** bit keeps track of whether or not a block has been modified while it is in the cache. A slot that is modified must be written back to the main memory before the slot is reused for another block.

A referenced location that is found in the cache results in a **hit**, otherwise, the result is a **miss**.

ASSOCIATIVE MAPPED CACHE:



When a reference is made to a main memory address, the cache hardware intercepts the reference and searches the cache tag memory to see if the requested block is in the cache. For each slot, if the valid bit is 1, then the tag field of the referenced address is compared with the tag field of the slot. All of the tags are searched in parallel.

If any tag in the cache tag memory matches the tag field of the memory reference, then the word is taken from the position in the slot specified by the word field. If the referenced word is not found in the cache, then the main memory block that contains the word is brought into the cache and the referenced word is then taken from the cache. The tag, valid and dirty fields are updated, and the program resumes execution.

When all of the valid bits are 1, one of the slots must be freed for a new block by any replacement policy:

★ **Least Recently Used (LRU)**: A time stamp is added to each slot, which is updated when any slot is accessed. The last recently used slot is discarded when a slot must be freed for a new block.

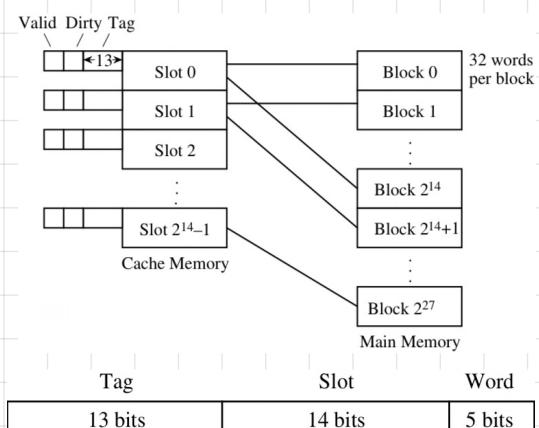
★ **First In- First Out (FIFO)**: Replaces slots one after the next in the order of their physical locations in the cache.

★ **Least Frequently Used (LFU)**: The least frequently used slot is freed. Only one slot is updated at a time by incrementing a frequency counter that is attached to each slot.

★ **Random**: Chooses a slot at random.

The **LFU** is only slightly better than the **Random** policy. The **LRU** can be implemented efficiently.

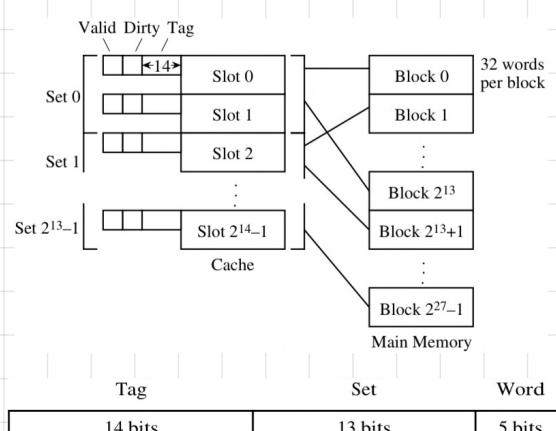
DIRECT MAPPED CACHE:



Each slot corresponds to an explicit set of main memory blocks. When a reference is made to a main memory address, the slot fields identifies in which of all the slots the block will be found if its in the cache. If the valid bit is 1, then the tag field of the referenced address is compared with the tag field of the slot. If the tag fields are the same, then the word is taken from the position in the slot specified by the word field.

If the valid bit is 1 but the tag fields are not the same, then the slot is written back to main memory if the dirty bit is set, and the corresponding main memory block is then read into the slot. For a program that has just started execution, the valid bit will be 0 and so the block is simply written to the slot. The valid bit for the block is then set to 1, and the program resumes execution.

SET ASSOCIATIVE MAPPED CACHE:



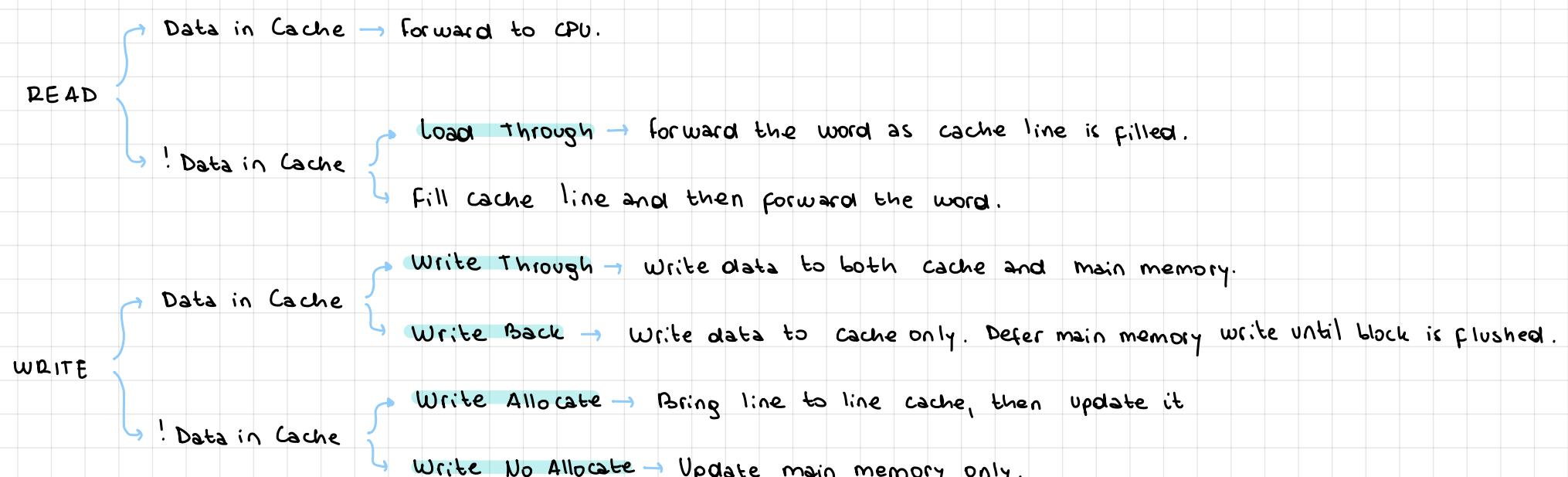
Combines the simplicity of direct mapping with the flexibility of associative mapping. The associative portion is limited to just a few slots that make up a set.

If n blocks make up a set, it is a n -way set associative cache.

When an address is mapped to a set, the direct mapping scheme is used, and then associative mapping is used within a set.

The tag memory increases only slightly from the direct mapping and only two tags need to be searched for each memory reference.

CACHE PERFORMANCE:



- SPLIT CACHE: Instructions and data are stored in separate sections of memory. Since instructions slots can never be dirty, an instruction cache is simpler than a data cache. Most of the memory traffic moves away from main memory rather than toward it. Instructions in an executing program are only read from the main memory, and are never written to the memory except by the system loader. Operations on data typically involve two operands and storing a single result.

MULTI LEVEL CACHES

The fastest level (L1) is on the same chip as the processor, and the remaining cache is placed off.