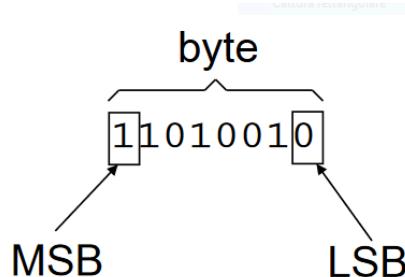


ARCHITETTURA DEGLI ELABORATORI

Rappresentazione dell'informazione

Introduzione

- Una rappresentazione è un modo per descrivere un'entità
- Proprietà principali di un sistema numerico posizionale:
 - Ogni numero intero può essere rappresentato (range illimitato)
 - A ogni numero intero corrisponde un solo insieme ordinato di cifre (rappresentazione unica)
- **Byte:** una sequenza di otto bit consecutivi



- **Most Significant Bit (MSB):** il bit più a sinistra
- **Least Significant Bit (LSB):** il bit più a destra
- La rappresentabilità dei valori è legata al numero di cifre disponibili
- Il numero di cifre impiegate nella rappresentazione di valori numerici è limitato
- Si ha **overflow** quando si è nell'impossibilità di rappresentare il risultato di una operazione con il numero di cifre a disposizione

Numeri Interi

- **MODULO E SEGNO**
 - Aggiungo a sinistra del numero binario un bit per il segno
 - 0 → POSITIVO
 - 1 → NEGATIVO
- **COMPLEMENTO A 1**
 - Nego tutte le cifre
- **COMPLEMENTO A 2**
 - Complemento a 1 → Aggiungo +1
- **ECCESSO 128**
 - Al numero da convertire aggiungo 128, poi trasformo in binario

N decimal	N binary	-N signed mag.	-N 1's compl.	-N 2's compl.	-N excess 128
1	00000001	10000001	11111110	11111111	01111111
2	00000010	10000010	11111101	11111110	01111110
3	00000011	10000011	11111100	11111101	01111101
4	00000100	10000100	11111011	11111100	01111100
5	00000101	10000101	11111010	11111011	01111011
6	00000110	10000110	11111001	11111010	01111010
7	00000111	10000111	11111000	11111001	01111001
8	00001000	10001000	11110111	11111000	01111000
9	00001001	10001001	11110110	11110111	01110111
10	00001010	10001010	11110101	11110110	01110110
20	00010100	10010100	11101011	11101100	01101100
30	00011110	10011110	11100001	11100010	01100010
40	00101000	10101000	11010111	11011000	01011000
50	00110010	10110010	11001101	11001110	01001110
60	00111100	10111100	11000011	11000100	01000100
70	01000110	11000110	10111001	10111010	00111010
80	01010000	11010000	10101111	10110000	00110000
90	01011010	11011010	10100101	10100110	00100110
100	01100100	11100100	10011011	10011100	00011100
127	01111111	11111111	10000000	10000001	00000001
128	Nonexistent	Nonexistent	Nonexistent	10000000	00000000

Figure A-7. Negative 8-bit numbers in four systems.

NB: In complemento a 1, il riporto più a sinistra viene sommato alla fine, mentre in complemento a 2 viene SCARTATO

Numeri razionali

- Con un numero finito di cifre è solo possibile rappresentare un **numero razionale che approssima con un certo errore il numero reale dato**
- Vengono usate due notazioni:
 - Notazione in virgola fissa**
 - Dedica parte delle cifre alla parte intera e le altre alla parte frazionaria parte frazionaria: $\pm XXX.YY$
 - Il numero di bit della parte frazionaria e quello della parte intera sono **costanti**
 - Per le operazioni aritmetiche bisogna **incolonnare virgola sotto virgola** gli addendi
 - I **numeri negativi** possono essere rappresentati in MS o CA2
 - Notazione in virgola mobile**
 - Dedica alcune cifre a rappresentare un esponente della base che indica l'ordine di grandezza del numero rappresentato
 - La posizione della virgola non è fissa, ma varia per avere una rappresentazione in notazione scientifica in cui:
 - Un'unica cifra a sinistra della virgola
 - Una parte frazionaria una parte frazionaria
 - Un esponente al quale si deve elevare la base del numero

- Semplice/singola precisione su 32 bit (notazione MS):

1	8	23
SEGNO	ESPONENTE	MANTISSA

- Doppia precisione su 64 bit (notazione MS):

1	11	52
SEGNO	ESPONENTE	MANTISSA

- L'esponente può assumere valori negativi (quindi i numeri in virgola mobile possono essere rappresentati in MS e CA2)
- ESPONENTE IN ECCESSO 127**

Tabella ASCII

- Codice su 7 bit
- Usato per la codifica dei caratteri
- Assegna un codice univoco a un carattere

ASCII Estesa

- Codice su 8 bit

Dec	Hex	Oct	Bin	Char	Dec	Hex	Oct	Bin	Char	Dec	Hex	Oct	Bin	Char	Dec	Hex	Oct	Bin	Char
0	0x00	000	0000000	NUL	32	0x20	040	0100000	space	64	0x40	100	1000000	@	96	0x60	140	1100000	'
1	0x01	001	0000001	SOH	33	0x21	041	0100001	!	65	0x41	101	1000001	A	97	0x61	141	1100001	a
2	0x02	002	0000010	STX	34	0x22	042	0100010	"	66	0x42	102	1000010	B	98	0x62	142	1100010	b
3	0x03	003	0000011	ETX	35	0x23	043	0100011	#	67	0x43	103	1000011	C	99	0x63	143	1100011	c
4	0x04	004	0000100	EOT	36	0x24	044	0100100	\$	68	0x44	104	1000100	D	100	0x64	144	1100100	d
5	0x05	005	0000101	ENQ	37	0x25	045	0100101	%	69	0x45	105	1000101	E	101	0x65	145	1100101	e
6	0x06	006	0000110	ACK	38	0x26	046	0100110	&	70	0x46	106	1000110	F	102	0x66	146	1100110	f
7	0x07	007	0000111	BEL	39	0x27	047	0100111	'	71	0x47	107	1000111	G	103	0x67	147	1100111	g
8	0x08	010	0001000	BS	40	0x28	050	0101000	(72	0x48	110	1001000	H	104	0x68	150	1101000	h
9	0x09	011	0001001	TAB	41	0x29	051	0101001)	73	0x49	111	1001001	I	105	0x69	151	1101001	i
10	0x0A	012	0001010	LF	42	0x2A	052	0101010	*	74	0x4A	112	1001010	J	106	0x6A	152	1101010	j
11	0x0B	013	0001011	VT	43	0x2B	053	0101011	+	75	0x4B	113	1001011	K	107	0x6B	153	1101011	k
12	0x0C	014	0001100	FF	44	0x2C	054	0101100	,	76	0x4C	114	1001100	L	108	0x6C	154	1101100	l
13	0x0D	015	0001101	CR	45	0x2D	055	0101101	-	77	0x4D	115	1001101	M	109	0x6D	155	1101101	m
14	0x0E	016	0001110	SO	46	0x2E	056	0101110	.	78	0x4E	116	1001110	N	110	0x6E	156	1101110	n
15	0x0F	017	0001111	SI	47	0x2F	057	0101111	/	79	0x4F	117	1001111	O	111	0x6F	157	1101111	o
16	0x10	020	0010000	DLE	48	0x30	060	0110000	0	80	0x50	120	1010000	P	112	0x70	160	1110000	p
17	0x11	021	0010001	DC1	49	0x31	061	0110001	1	81	0x51	121	1010001	Q	113	0x71	161	1110001	q
18	0x12	022	0010010	DC2	50	0x32	062	0110010	2	82	0x52	122	1010010	R	114	0x72	162	1110010	r
19	0x13	023	0010011	DC3	51	0x33	063	0110011	3	83	0x53	123	1010011	S	115	0x73	163	1110011	s
20	0x14	024	0010100	DC4	52	0x34	064	0110100	4	84	0x54	124	1010100	T	116	0x74	164	1110100	t
21	0x15	025	0010101	NAK	53	0x35	065	0110101	5	85	0x55	125	1010101	U	117	0x75	165	1110101	u
22	0x16	026	0010110	SYN	54	0x36	066	0110110	6	86	0x56	126	1010110	V	118	0x76	166	1110110	v
23	0x17	027	0010111	ETB	55	0x37	067	0110111	7	87	0x57	127	1010111	W	119	0x77	167	1110111	w
24	0x18	030	0011000	CAN	56	0x38	070	0111000	8	88	0x58	130	1011000	X	120	0x78	170	1111000	x
25	0x19	031	0011001	EM	57	0x39	071	0111001	9	89	0x59	131	1011001	Y	121	0x79	171	1111001	y
26	0x1A	032	0011010	SUB	58	0x3A	072	0111010	:	90	0x5A	132	1011010	Z	122	0x7A	172	1111010	z
27	0x1B	033	0011011	ESC	59	0x3B	073	0111011	;	91	0x5B	133	1011011	[123	0x7B	173	1111011	{
28	0x1C	034	0011100	FS	60	0x3C	074	0111100	<	92	0x5C	134	1011100	\	124	0x7C	174	1111100	
29	0x1D	035	0011101	GS	61	0x3D	075	0111101	=	93	0x5D	135	1011101]	125	0x7D	175	1111101	}
30	0x1E	036	0011110	RS	62	0x3E	076	0111110	>	94	0x5E	136	1011110	^	126	0x7E	176	1111110	~
31	0x1F	037	0011111	US	63	0x3F	077	0111111	?	95	0x5F	137	1011111	_	127	0x7F	177	1111111	DEL

UNICODE

- Inizialmente rappresentato come una codifica su 16 bit, ma poi esteso a 24 e 32 bit a 24 e 32 bit
- Unicode è in continua evoluzione e continua ad aggiungere più caratteri

Reti logiche

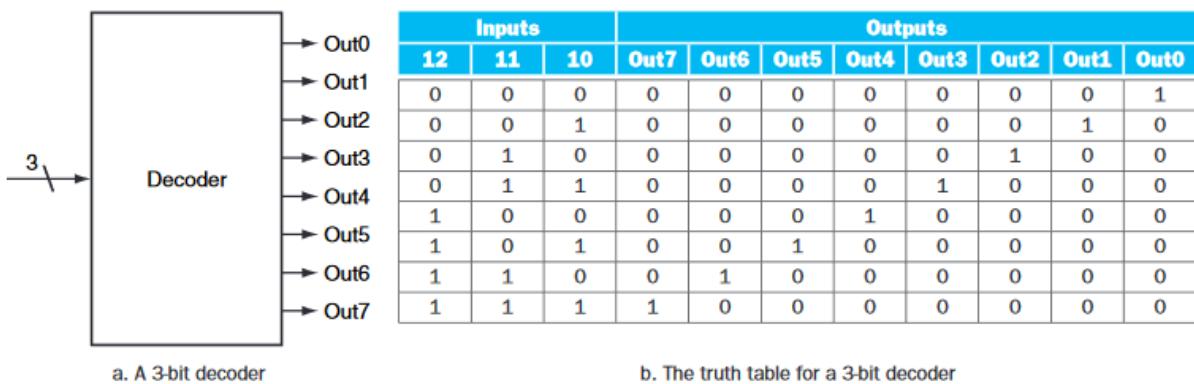
Introduzione

- **RETI COMBINATORIE**
 - Combinano gli ingressi
- **RETI SEQUENZIALI**
 - Combinano sia gli **INGRESSI PRECEDENTI** sia quelli **ATTUALI**

Reti combinatorie

Decoder

- Componente con n input e 2^n output



a. A 3-bit decoder

b. The truth table for a 3-bit decoder

Multiplexer

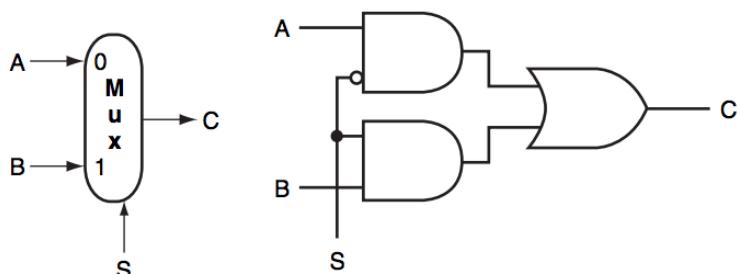


FIGURE C.3.2 A two-input multiplexor on the left and its implementation with gates on the right. The multiplexor has two data inputs (A and B), which are labeled 0 and 1, and one selector input (S), as well as an output C . Implementing multiplexors in Verilog requires a little more work, especially when they are wider than two inputs. We show how to do this beginning on page C-23.

- Componente con n input, 2^m selettori e 1 output

ROM

- Read-Only Memory
- “Memory” è fuorviante
- Ha **2^n indirizzi (altezza)** e **2^n output di larghezza variabile**
- Ne esistono diversi tipi
 - PROM → Programmable
 - EPROM → Erasable-Programmable
 - EEPROM → Electric-Erasable-Programmable

ALU (1-bit)

- Gestisce dati a 32 bit
- Modalità didattica → 32 ALU a 1 bit
- **PARTE LOGICA:**

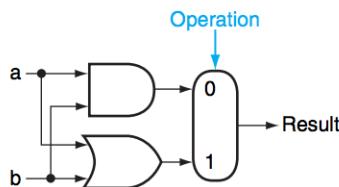


FIGURE C.5.1 The 1-bit logical unit for AND and OR.

- **PARTE ARITMETICA:**

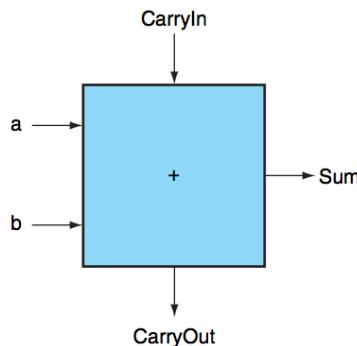


FIGURE C.5.2 A 1-bit adder. This adder is called a full adder; it is also called a (3,2) adder because it has 3 inputs and 2 outputs. An adder with only the a and b inputs is called a (2,2) adder or half-adder.

- SCHEMA COMPLETO:

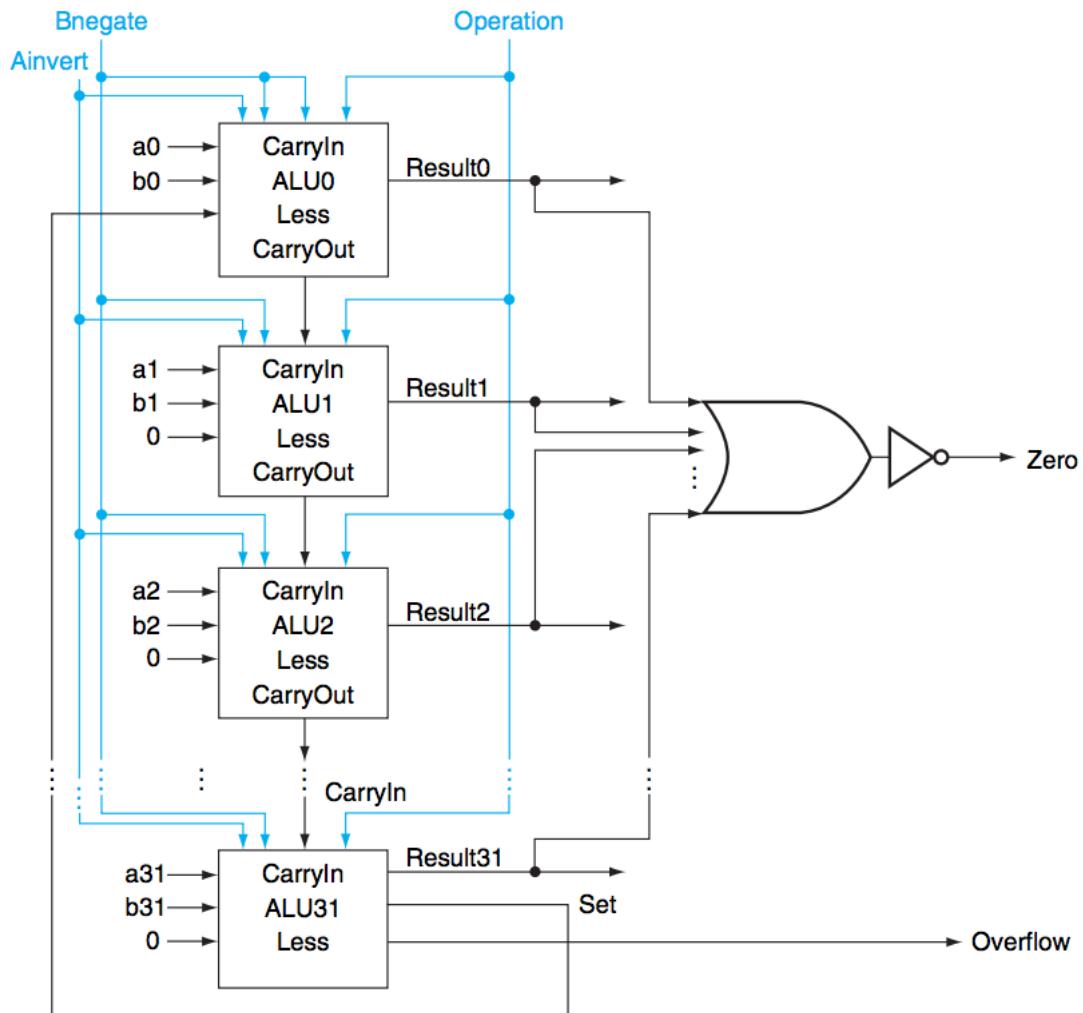


FIGURE C.5.12 The final 32-bit ALU. This adds a Zero detector to Figure C.5.11.

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

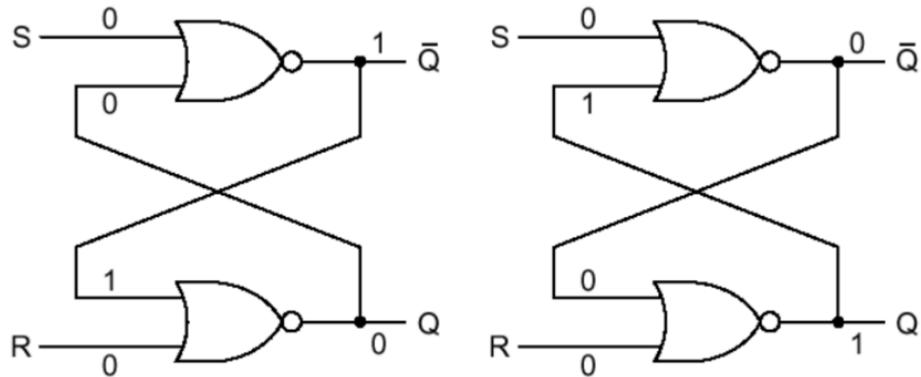
FIGURE C.5.13 The values of the three ALU control lines, Bnegate, and Operation, and the corresponding ALU operations.

Reti sequenziali

- Per realizzare circuiti sequenziali è necessario un elemento di memoria per memorizzare lo stato

SR Latch

- S-R Latch è composto da 2 porte NOR concatenate
- Costituisce l'elemento base per costruire elementi di memoria più complessi
- Memorizza 1 bit e ricorda i valori di input precedenti



- La combinazione $S=1$ e $R=1$ non deve essere mai presentata al latch perché sarebbe $Q = \text{NOT } Q = 0$, ma il valore può essere arbitrario in quanto dipende dall'ordine del resetting

Flip Flop D

- Usa metodologia edge-triggered

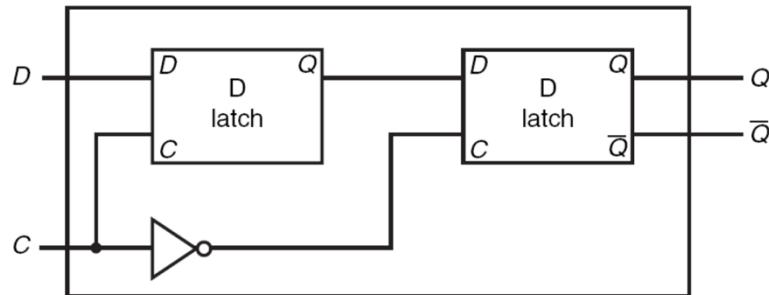
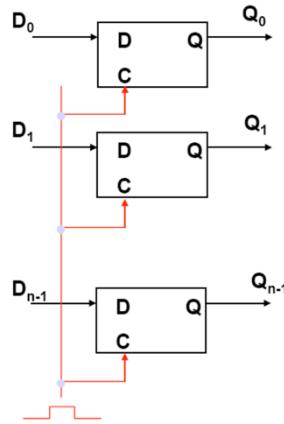


FIGURE C.8.4 A D flip-flop with a falling-edge trigger. The first latch, called the master, is open and follows the input D when the clock input, C , is asserted. When the clock input, C , falls, the first latch is closed, but the second latch, called the slave, is open and gets its input from the output of the master latch.

Register File

- Il **Datapath** (la parte operativa della CPU) contiene (oltre alla ALU) anche **registri** che **memorizzano** (all'interno della CPU) **gli operandi** delle istruzioni aritmetico/logiche
- Un registro è costituito da **n flip-flop** dove **n** è il **numero di bit di un registro**
 - Nel MIPS ogni registro è di 1 word = 4 byte = 32 bit
- I registri sono organizzati in un Register File
 - Il Register File del MIPS ha 32 registri ($32 \times 32 = 1024$ flip-flop)
 - Il Register File permette la lettura di 2 registri e la scrittura di 1 registro



- Nel Datapath della CPU il clock non entra direttamente nei vari flip-flop; viene messo in AND con un segnale di controllo "Write"
- Il segnale Write (in AND con il clock) determina se, in corrispondenza del fronte di discesa del clock, il valore D debba (o meno) essere memorizzato nel registro

Lettura da Register file

- Utilizza 2 segnali che indicano i registri da leggere (Read Reg1, Read Reg2)
- Utilizza 2 multiplexer: ognuno con 32 ingressi e un segnale di controllo da 5 bit (Read Reg1, Read Reg2)
 - Osservazione: Il register file fornisce sempre in output una coppia di registri; se i segnali di read non sono significativi tali registri vanno ignorati

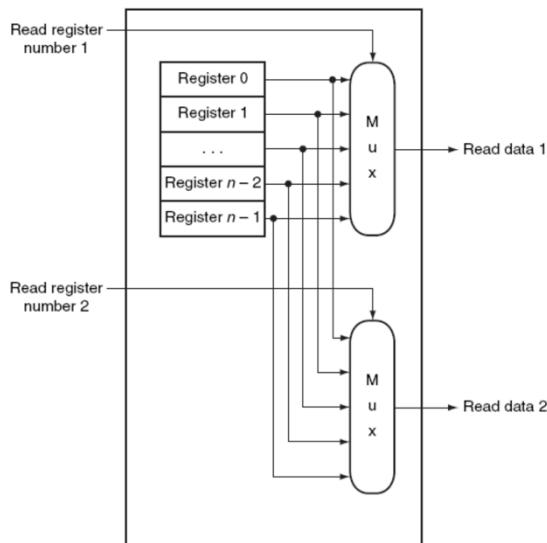


FIGURE C.8.8 The implementation of two read ports for a register file with n registers can be done with a pair of n -to-1 multiplexors, each 32 bits wide. The register read number signal is used as the multiplexer selector signal. Figure C.8.9 shows how the write port is implemented.

Scrittura nel Register File

- Utilizza 3 segnali:
 - Il registro da scrivere (Register Number)
 - Il valore da scrivere (nel Register Data)
 - Il segnale di controllo Write che abilita la scrittura nel Register File
- Utilizza un decoder che decodifica il numero del registro da scrivere
- Il segnale Write (già in AND con il clock) è in AND con l'output del decoder
- Se Write non è affermato nessun valore sarà scritto nel registro

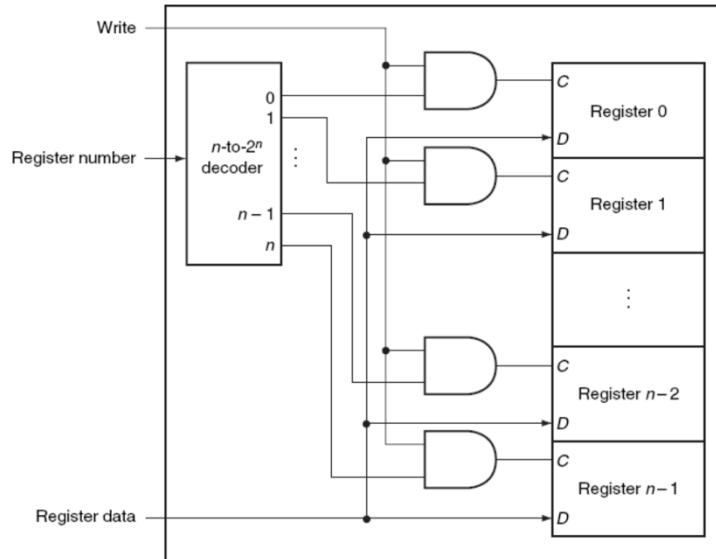
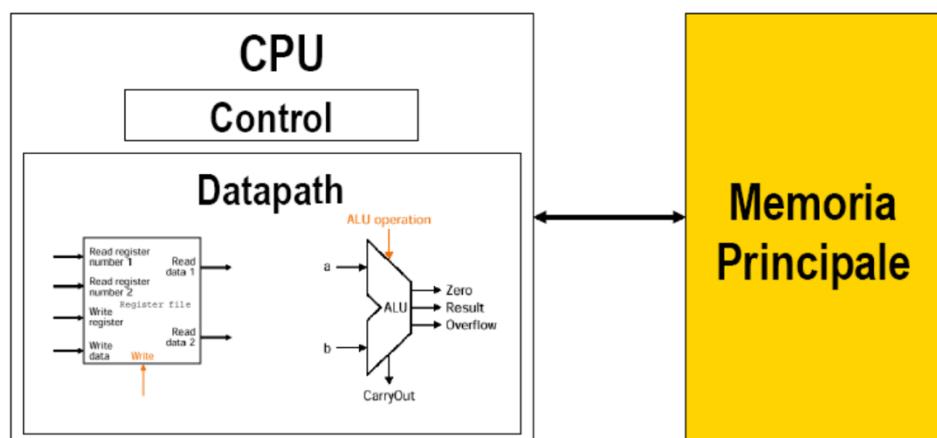


FIGURE C.8.9 The write port for a register file is implemented with a decoder that is used with the write signal to generate the C input to the registers. All three inputs (the register number, the data, and the write signal) will have setup and hold-time constraints that ensure that the correct data is written into the register file.

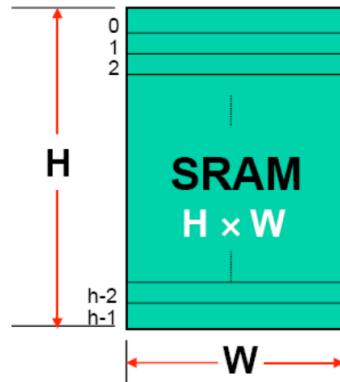
- Cosa succede se uno stesso registro del Register File viene acceduto in lettura e scrittura durante lo stesso ciclo di clock?
 - Sarà letto il valore memorizzato nel ciclo di clock precedente, mentre sarà scritto in nuovo valore indicato perché la scrittura sul registro avviene sul fronte di discesa del clock

Memoria Centrale

- Per memorizzare i dati è necessaria una mole di memoria molto superiore
- Si ricorre all'utilizzo di **RAM**
 - **Meno veloce** dei registri
 - **Molto più capiente**

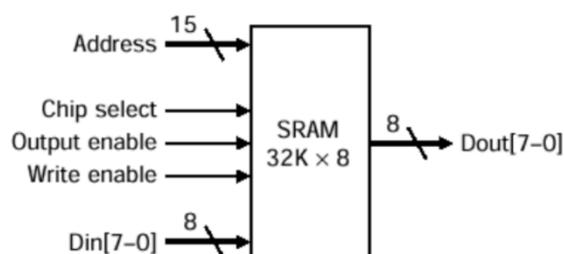


- **SRAM:** Static Ram
 - Usata per costruire memorie più veloci come la cache
 - Tempi di accesso: 0.5 – 2.5 ns
 - Realizzata come **matrice di latch**
 - Larghezza o ampiezza W (numero di latch per ogni cella)
 - Altezza H (numero di celle indirizzabili)
 - Singolo indirizzo usato sia per lettura che per scrittura
 - Non è possibile leggere e scrivere contemporaneamente (come per Register File)



**Numero di bit
dell'indirizzo: $\log_2 H$**

- ESEMPIO 1:
 - 32K x 8 (32 celle da 8 bit → 256Kb)
 - 2^{18} linee di indirizzo
 - 8 linee di output
- Peculiarità:
 - **Non si possono usare multiplexer e decoder come per Register File**
 - Con un numero elevato di celle di memoria avremmo bisogno di decoder e multiplexer **enormi**



- **DRAM:** Dynamic Ram
 - Ogni bit è memorizzato tramite un condensatore (non usa latch)
 - Usata per memorie capienti come quella principale
 - Tempi di accesso: 50-70 ns
 - È necessario rinfrescare il contenuto della DRAM a intervalli di tempo prefissati
 - **MENO COSTOSA, PIÙ CAPIENTE, PIÙ LENTA DELLE SRAM**
 - ESEMPIO:
 - DRAM di $4M \times 1 = 4Mb$
 - Indirizzo totale su 22 bit
 - Indirizzo spezzato in 2 pezzi da 11 bit ciascuno
 - Parte alta e parte bassa dell'indirizzo considerate come indirizzo di riga e di colonna:
 - Indirizzo di riga ha effetto sul decoder
 - Indirizzo di colonna ha effetto sul multiplexer

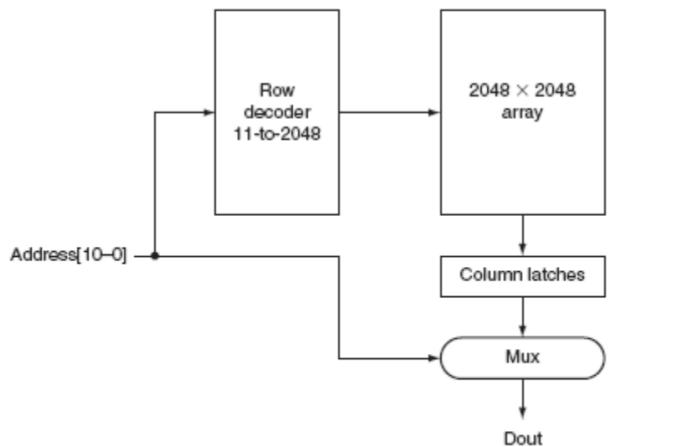
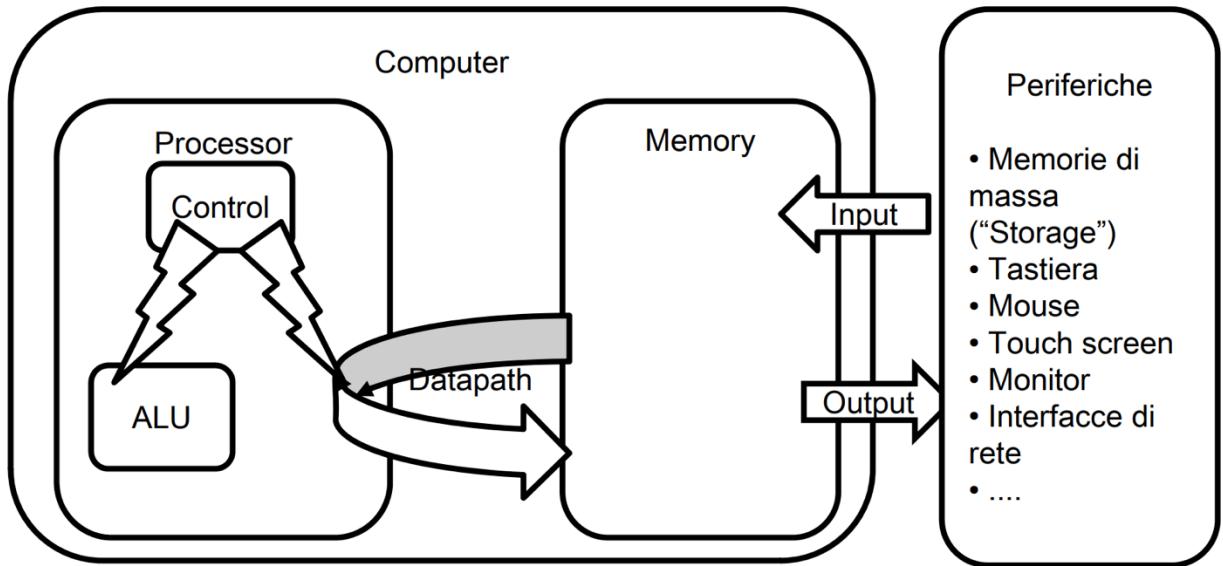


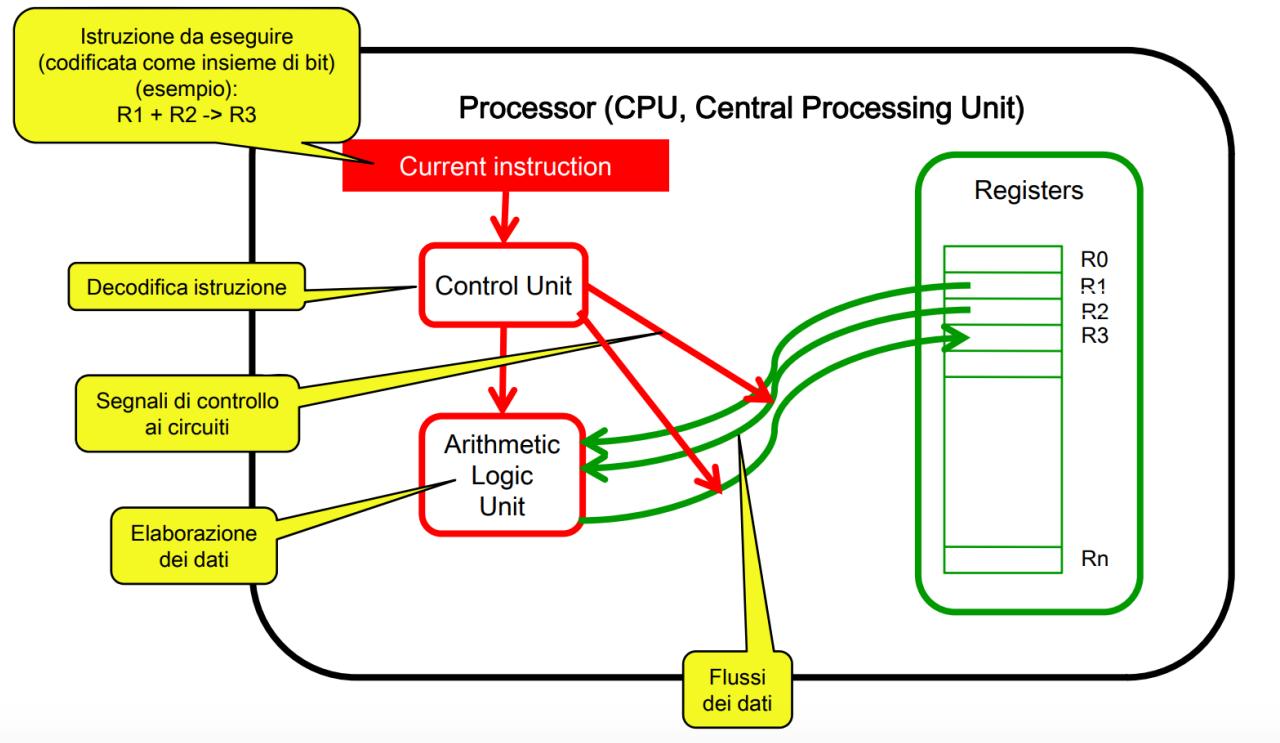
FIGURE C.9.6 A $4M \times 1$ DRAM is built with a 2048×2048 array. The row access uses 11 bits to select a row, which is then latched in 2048 1-bit latches. A multiplexor chooses the output bit from these 2048 latches. The RAS and CAS signals control whether the address lines are sent to the row decoder or column multiplexor.

ISA (Instruction Set Architecture)

The “big picture”



Processore: Istruzione base sui registri



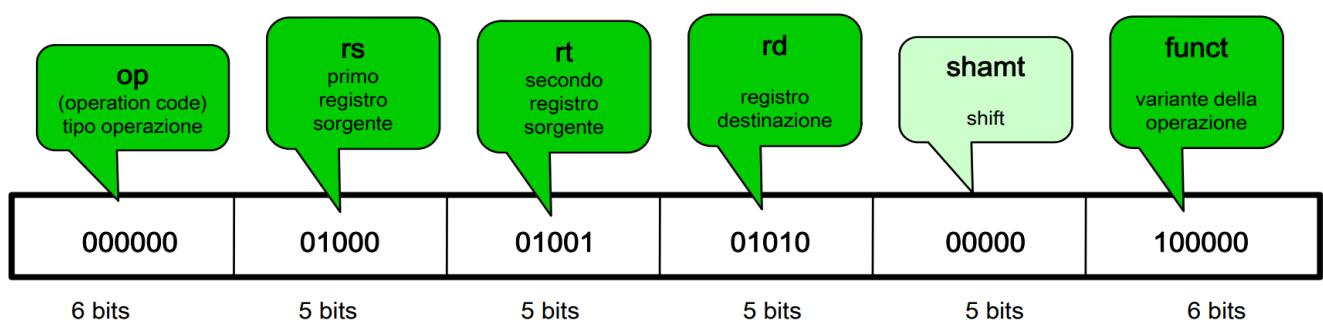
Tipologie progetto CPU

- **RISC** → Reduced Instruction Set Computing (es. MIPS)
- **CISC** → Complex Instruction Set Computing

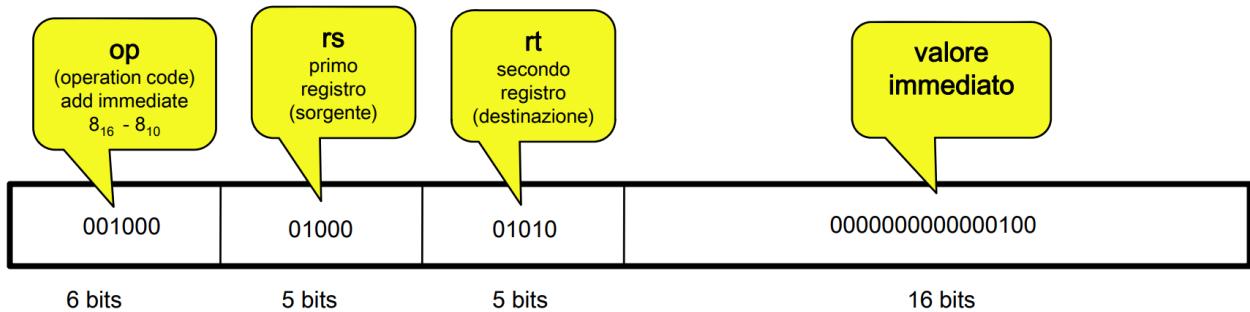
Architettura MIPS

- **RISC**
- **32 Registri di 32 bit**
- **Istruzioni di 32 bit (4 byte)**
 - Manipolazione dati SOLO sui registri
 - Trasferimento dati tra memoria e registri
 - Alterazione del flusso di controllo (Jumps)
- Problema architetturale
 - Rappresentazione di tutte le operazioni in 32 bit
- **TIPI DI ISTRUZIONE**
 - **R-TYPE**
 - **I-TYPE**
 - **J-TYPE**

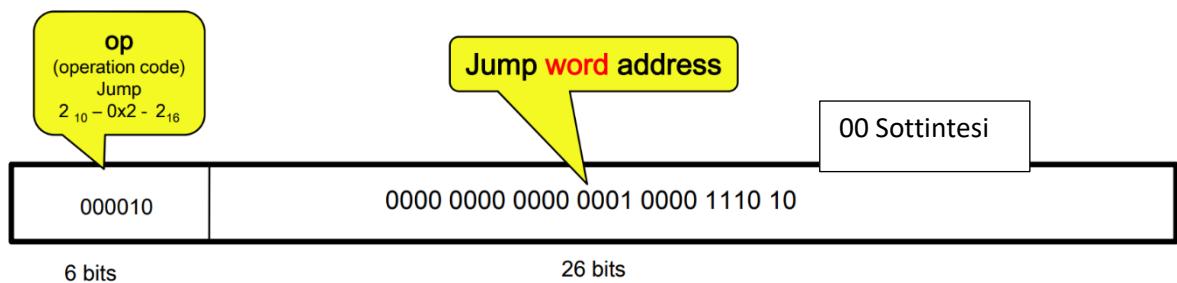
Istruzioni R-TYPE



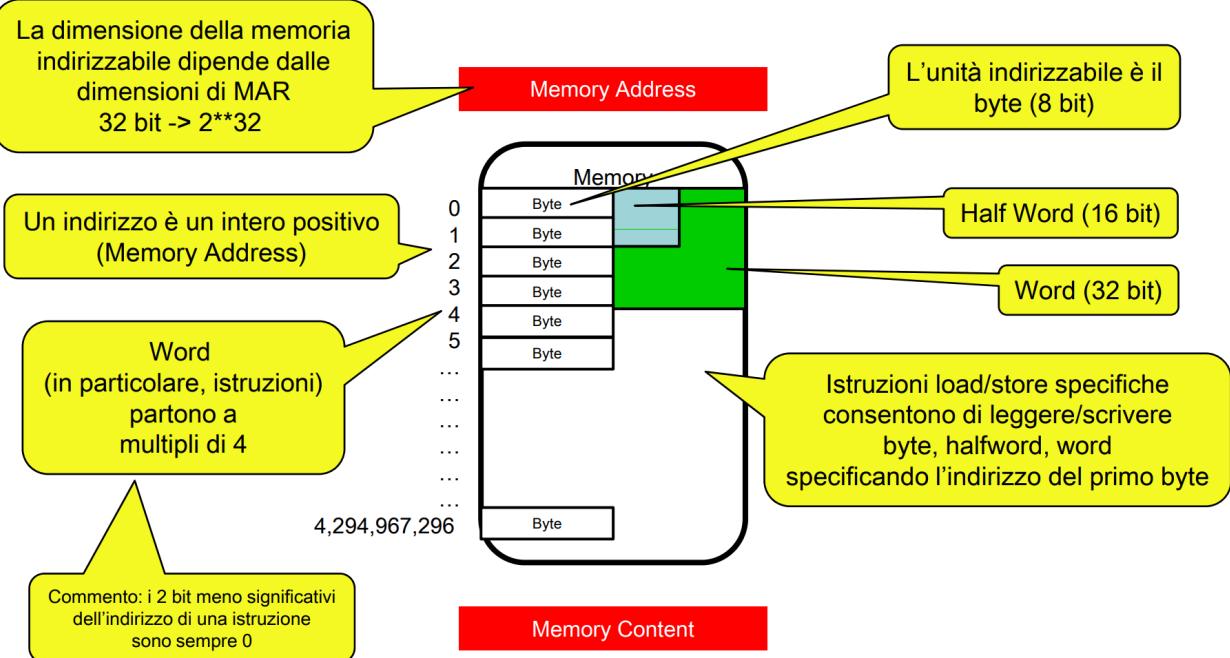
Istruzioni I-TYPE



Istruzioni J-TYPE



Indirizzamento della memoria



Tipi di addressing

- Immediate → I-TYPE
- Register → R-TYPE
- Base → I-TYPE (Registro base + Offset)
- PC-Relative → I-TYPE (PC + Offset)
- Pseudodirect → J-TYPE

NB: L'OFFSET DELLE OPERAZIONI LW, SW ecc. CORRISPONDE AI BYTE

```
lw $t0, 56($s0)
add $t1, $s1, $t0
sw $t1, 68($s0)
```

(3 parole più avanti)

Branches

- Finestra → $-2^{15} \leq BA < 2^{15}$
 - Viene usato il PC come "base" da cui eseguire l'offset
 - **NB:** Se CA è l'indirizzo dell'istruzione corrente, si salta a **CA + 4 + BranchAddr*4**, dove BranchAddr corrisponde al numero di operazioni da saltare (ramo false del branch)

Linguaggi

Caratteristiche

MACCHINA	ASSEMBLY	ALTO LIVELLO
<ul style="list-style-type: none">• Direttamente comprensibile• Programmazione più lunga• Facile commettere errori	<ul style="list-style-type: none">• Rappresentazione simbolica• Più comprensibile• Tradotto dall'assemblatore in linguaggio macchina	<ul style="list-style-type: none">• Tradotti dal compilatore in linguaggio assembler• Simile al linguaggio corrente• Incremento di produttività• Portabilità

Catena programmativa

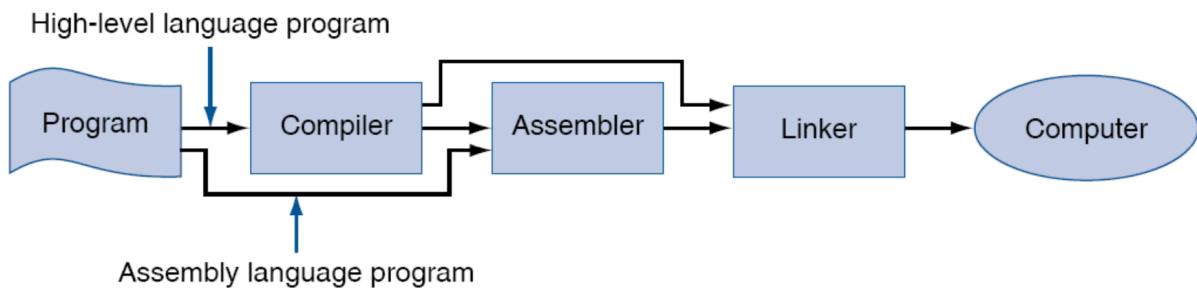


FIGURE A.1.6 Assembly language either is written by a programmer or is the output of a compiler.

Compilatore

- Ha il compito di tradurre un programma scritto in linguaggio ad alto livello in linguaggio assembler
- Spesso con il termine compilazione si indica l'intero processo di traduzione da linguaggio ad alto livello a linguaggio macchina (essendo ad alto livello a linguaggio macchina (essendo l'assemblatore spesso integrato con il compilatore)

Assemblatore

- **Converte** un programma assembler (file sorgente) in linguaggio macchina (file oggetto)
- Gestisce le **etichette**
- Gestisce **pseudoistruzioni**
- Gestisce **numeri in base diverse**
- **Assemblaggio:**
 - Procedimento sequenziale di traduzione
 - Applicato modulo per modulo, costruendo la **tabella dei simboli**
 - Traduce i **codici simbolici in istruzioni**
 - Traduce i **riferimenti simbolici in indirizzi**
- **DEVE LEGGERE IL PROGRAMMA DUE VOLTE PER VIA DEI RIFERIMENTI DELLE ETICHETTE → TRADUTTORE A DUE PASSI**

- Ogni modulo assemblato di default parte da 0
- **TABELLA DEI SIMBOLI:**
 - **Primo passo** → Contiene i riferimenti simbolici del modulo e i relativi indirizzi numerici
 - Per le **costanti** viene creata la coppia **<etichetta, valore>**
 - Per le etichette che definiscono **variabili** viene riservato lo spazio relativo creando la coppia **<etichetta, indirizzo>**
 - Per le etichette che costituiscono le istruzioni di jump l'assemblatore deve generare un riferimento all'indirizzo di jump
 - Global/Local → Scope di visibilità dei moduli
 - **NB:** Le etichette esterne **NON** vengono risolte dall'assembler

Linker

- Inserisce in memoria in modo simbolico il codice e i moduli dati
- Determini gli indirizzi dei dati e delle etichette che compaiono nelle istruzioni
- Corregge i riferimenti esterni e interni e **risolve i riferimenti in sospeso**
- **GENERA IL FILE ESEGUIBILE**

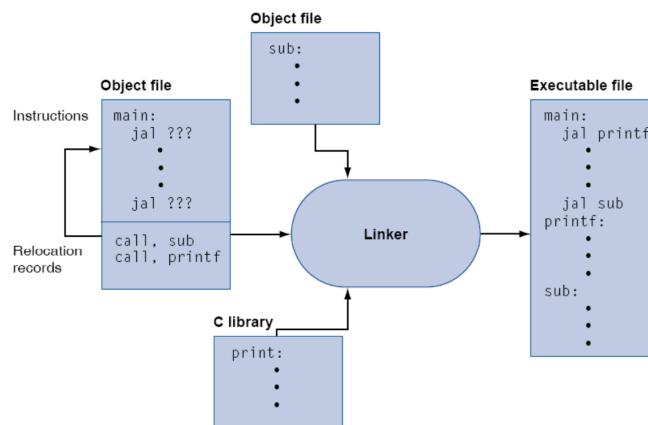


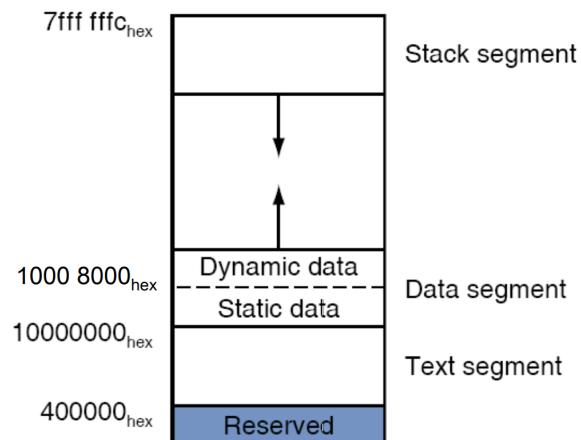
FIGURE A.3.1. The linker searches a collection of object files and program libraries to find nonlocal routines used in a program, combines them into a single executable file, and resolves references between routines in different files.

Loader

- **Lettura** dell'intestazione del file eseguibile per determinare la lunghezza di **Text Segment** e **Data Segment**
- **Crea** uno spazio di indirizzamento sufficiente a contenerli
- **Copia** delle istruzioni e dati dal file eseguibile all'interno della memoria
- **Copia** nello stack degli eventuali parametri passati al programma principale
- **Inizializzazione** dei registri e **impostazione** dello **stack pointer** affinché punti alla prima locazione libera
- **Salto a una procedura di startup** la quale copia i parametri nei registri argomento e chiama la procedura principale del programma
- Quando la procedura principale restituisce il controllo, la procedura di startup termina il programma con una chiamata alla funzione di sistema **exit**

Direttive

- NON corrispondono a istruzioni macchina
- Sono indicazioni date all'assembler per consentirgli di:
 - associare etichette simboliche a indirizzi
 - allocare spazio di memoria per le variabili
 - decidere in quali zone di memoria allocare istruzioni e dati



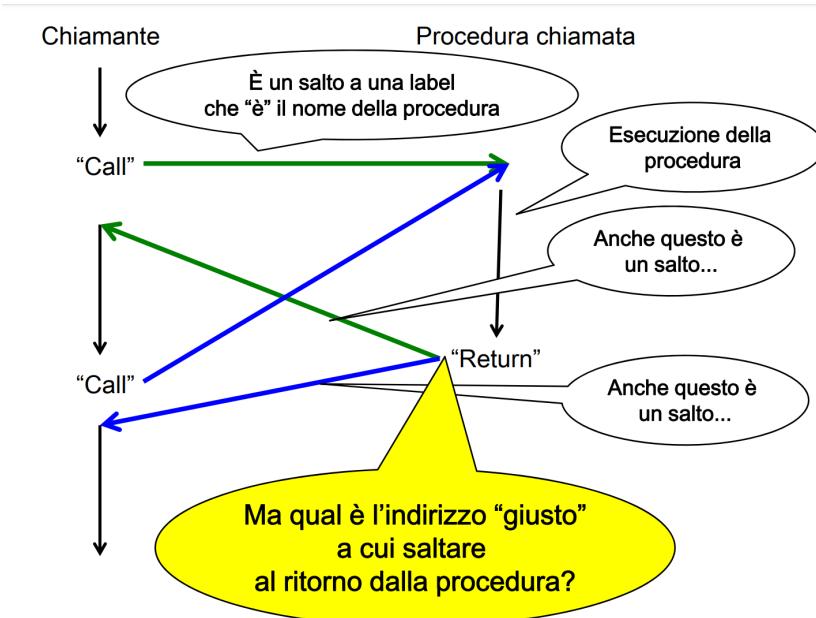
Pseudoistruzioni

- Non corrisponde a una sola istruzione macchina
- Tradotte dall'assembler in più istruzioni

Osservazioni

- I programmi sono immagazzinati in memoria insieme ai dati → **PROGRAMMA MEMORIZZATO**

Flusso di controllo



Istruzione JAL

- **Salta a una procedura** indicata nell'istruzione e **crea un collegamento a dove deve ritornare per continuare** l'esecuzione del chiamante
- **Salva** nel registro **\$ra** (registro 31) ("return address") l'indirizzo a cui tornare dopo l'esecuzione della procedura (è l'indirizzo successivo a quello dell'istruzione jal, cioè l'indirizzo in cui si trova la jal + 4)
- Tale indirizzo si trova nel registro PC (Program Counter)

Istruzione JR

- **Salta all'indirizzo contenuto in un registro**
- **Caso speciale** → jr \$ra → Serve per effettuare il ritorno da una procedura all'indirizzo salvato precedentemente in un jal

Usi convenzionali dei registri

- \$a0 - \$a3 → Argomenti procedure
- \$v0 - \$v1 → Registri valore per la restituzione di risultati
- Le convenzioni permettono di scrivere procedure che:
 - Possono essere scritte senza bisogno di sapere come è fatto il programma che le chiama che le chiama
 - Possono essere chiamate senza bisogno di sapere come sono fatte "dentro"
- **NB: UN PARAMETRO PUÒ ESSERE UTILIZZATO SIA COME VALORE CHE COME INDIRIZZO**

Osservazioni

- **Una procedura è un modo per organizzare in modo comprensibile e riutilizzabile il codice**
- I 6 passi di una procedura:
 - **Setting dei parametri** in un luogo accessibile alla procedura
 - **Trasferire il controllo** alla procedura
 - **Acquisire risorse** per l'esecuzione della procedura
 - **Eseguire** il compito richiesto
 - **Mettere il risultato in un luogo accessibile al chiamante**
 - **Restituire il controllo al punto di partenza**
- Cosa succede se una procedura ne chiama un'altra?
 - Si perde il contenuto di \$ra della prima chiamata?
- Procedure ricorsive?
 - -> uso dello stack
- Se una procedura usa registri, cosa succede del contenuto lasciato nei registri dal chiamante?
 - Convenzioni: registri \$s e \$t
- Dove stanno le variabili locali della procedura?
 - Stack frame

Syscall

- Analogo a una chiamata a procedura
- Convenzioni per le syscall:
 - Tabella a pag. A43 (Appendice A)
 - Impostare nel registro \$v0 il codice della chiamata
 - Impostare i parametri nei registri \$a0-\$a3 (come da tabella)

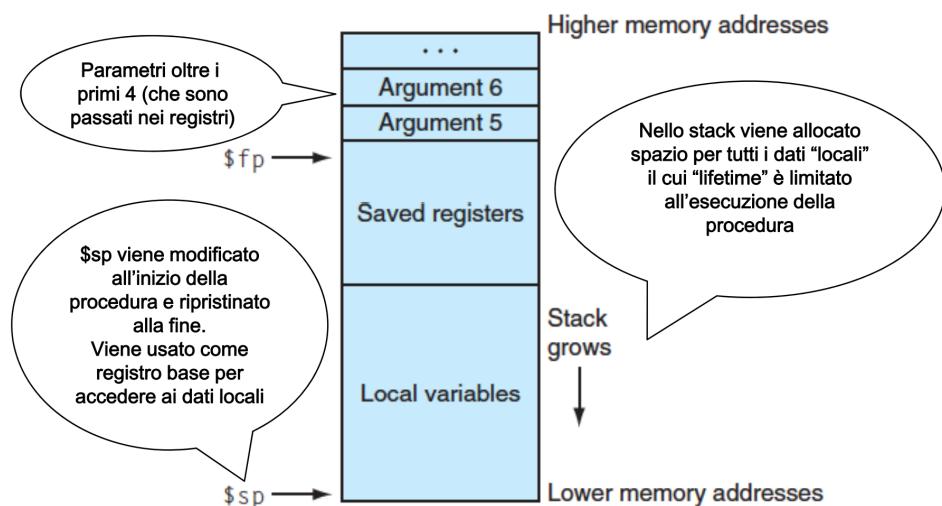
Registri salvati e temporanei

- REGISTRI SAVED (\$s)
 - Il chiamante **ha il diritto di aspettarsi che i registri saved siano immutati dopo la procedura chiamata**
 - Se la procedura chiamata ne fa utilizzo **deve ripristinarne il loro valore originale**
 - Dove salvare il contenuto dei registri \$s
 - Uso dello stack
- REGISTRI TEMPORANEI (\$t)
 - Il chiamante **non si può aspettare di trovare immutati i contenuti dei registri \$t dopo una chiamata a procedura**
 - I contenuti dei registri \$t **devono essere salvati dal chiamante** prima della chiamata a procedura

Procedure innestate

- Procedure “foglia” e “non foglia”
 - Una procedura foglia NON chiama altre procedure
 - Una procedura non foglia chiama altre procedure
- Cosa succede se una procedura ne chiama un’altra?
 - Si perde il contenuto di \$ra della prima chiamata?
 - Procedure ricorsive?
 - Bisogna che una procedura “non foglia” salvi il contenuto di \$ra e lo ripristini prima del ritorno

Uso dello Stack



Cosa fa una procedura chiamata

- Alloca spazio nello stack
- Decrementa \$sp per lasciare in stack lo spazio necessario al salvataggio (1 word per ciascun registro da salvare) (ricordare che lo stack cresce “verso il basso”)
- Salva \$ra
- Salva eventuali altri registri usando \$sp come registro base
- Ripristina i registri
- Incrementa \$sp per riportarlo alla situazione iniziale
- Jr \$ra (ritorno dalla procedura)

Cosa fa una procedura chiamante

- Impostare gli argomenti da passare alla procedura in \$a0-\$a3 (eventuali altri argomenti sono nella memoria o nello stack)
- Salvare eventualmente i registri \$a0-\$a3 e \$t0-\$t9 in quanto la procedura chiamata può usare liberamente questi registri
- Chiamare la procedura tramite l’istruzione jal nome_procedura

Datapath

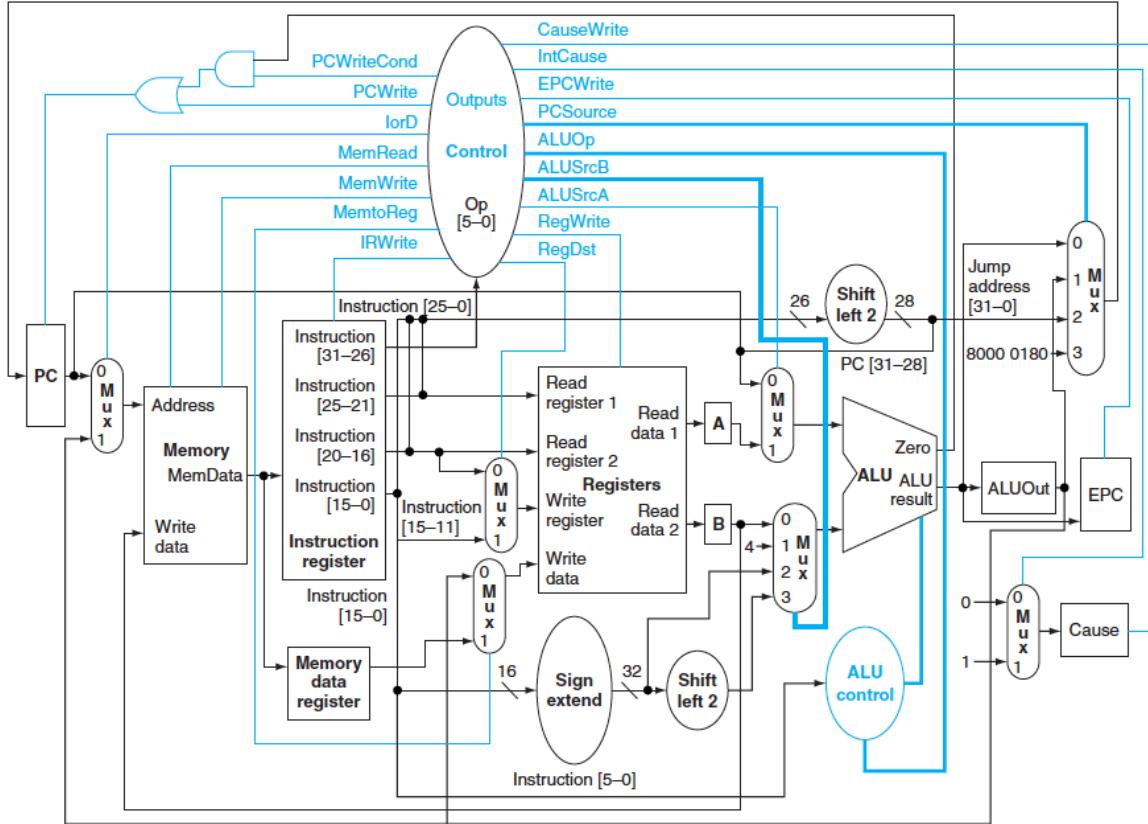
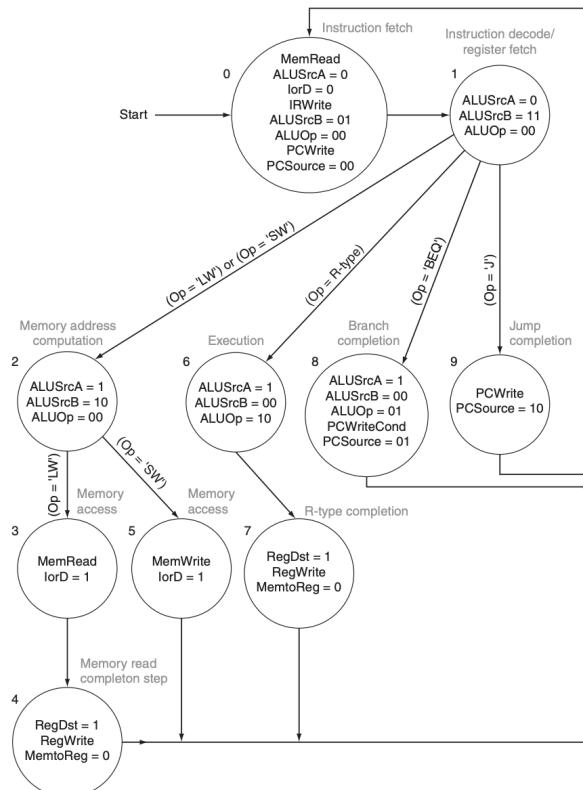


FIGURE 5.39 The multicycle datapath with the addition needed to implement exceptions. The specific additions include the Cause



Introduzione

- Le performance di una macchina sono determinate da 3 fattori
 - Numero di istruzioni da eseguire
 - Durata di un ciclo di clock
 - CPI (Clock Cycles per Instruction)

Implementazione

Tutte le operazioni eseguono questi due primi passaggi

1. **FETCH →**
 - a. Il PC punta all'indirizzo di memoria che contiene l'istruzione da eseguire e la preleva
 - b. Scrive dentro l'IR il valore dell'istruzione prelevata dalla memoria
 - c. Incrementa il suo valore di 4, mediante operazione con l'ALU e riscrive il puntatore a istruzione successiva dentro sé
2. **DECODE (Calcolo del target address) →**
 - a. L'ALU somma l'indirizzo base del PC al campo immediato/offset dell'istruzione (anche in caso di operazioni diverse da type-I) esteso di segno e shiftato di 2 verso sinistra
 - b. Vengono letti i registri (IR[25:21] e IR[20:16]) e memorizzati dentro A e B
 - c. Viene memorizzato il target address in ALUOut (anche se risultasse inutile utilizzarlo dopo)
3. **EXECUTE**
 - a. **R-TYPE**
 - i. **1° CICLO**
 1. $ALUOut \Leftarrow A \text{ op } B$ (sum, sub, and, or, slt)
 - ii. **2° CICLO**
 1. $IR[15:11] \Leftarrow ALUOut$
 - b. **JUMP**
 - i. $PC \Leftarrow IR[25:0] \ll 2$
 - c. **BEQ**
 - i. $ALUOut \Leftarrow A - B$
 - ii. If ($Zero == \text{true}$) $PC \Leftarrow \text{valore del target address calcolato nel DECODE}$
 - d. **SW**
 - i. $ALUOut \Leftarrow A + IR[15:0]$ SOLO esteso di segno
 - ii. $\text{Mem}[ALUOut] \Leftarrow B$
 - e. **LW**
 - i. $ALUOut \Leftarrow A + IR[15:0]$ SOLO esteso di segno
 - ii. $MDR \Leftarrow \text{Mem}[ALUOut]$
 - iii. $\text{RegFile}[IR[20:16]] \Leftarrow MDR$

Eccezioni

- **ECCEZIONI →** Sincrone all'esecuzione del programma (bad address, overflow)
- **INTERRUPT →** Asincrone
- **TRAPS (SysCalls) →** Programmatore

Il Mips32 prevede

- Registro Cause
- Gestore eccezioni → Software
- Registro EPC (Exception Program Counter) → Contiene l'istruzione da riprendere

Quando avviene un'eccezione:

1. Save
2. Cause identify
3. Salto al codice corrispondente la causa
4. Restore dei registri
5. Eret (torno all'istruzione scritta nell'EPC)

NB: Se quando sono nel gestore avviene un'altra eccezione non posso tornare all'indirizzo della seconda eccezione

- Metodo di **Vettorizzazione (gestione a livello Hardware)**

Causa + Indirizzo Base Vettore Eccezioni = Indirizzo del codice relativo all'eccezione

Mem[Indirizzo del codice relativo all'eccezione] → PC

Input/Output

Periferiche Mappate in Memoria

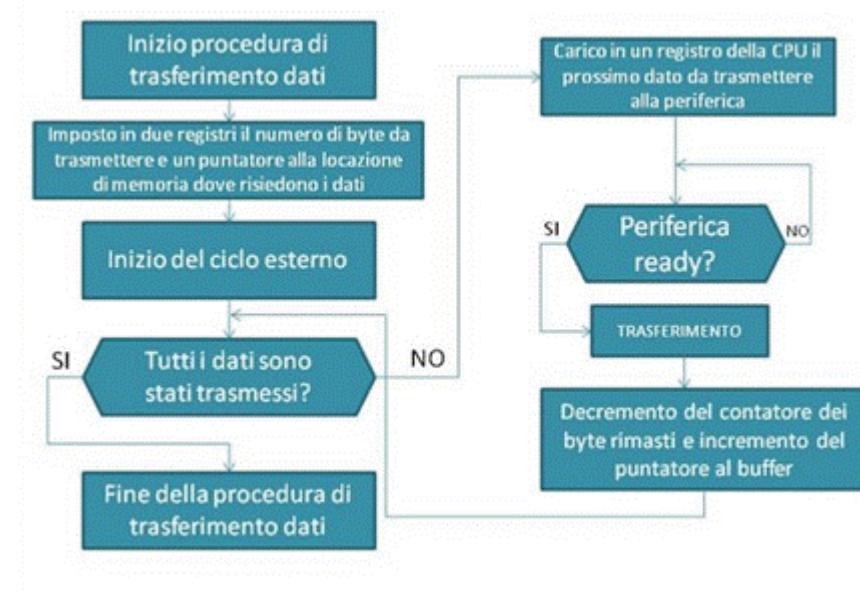
Ogni periferica viene riconosciuta dalla CPU come una locazione di memoria (lo spazio di indirizzamento della CPU comprende anche l'indirizzamento alle periferiche tramite BUS)

Ogni periferica possiede un'interfaccia di comunicazione con la CPU costituita da:

- Registro di stato
- Registro dati → I/O a seconda della periferica

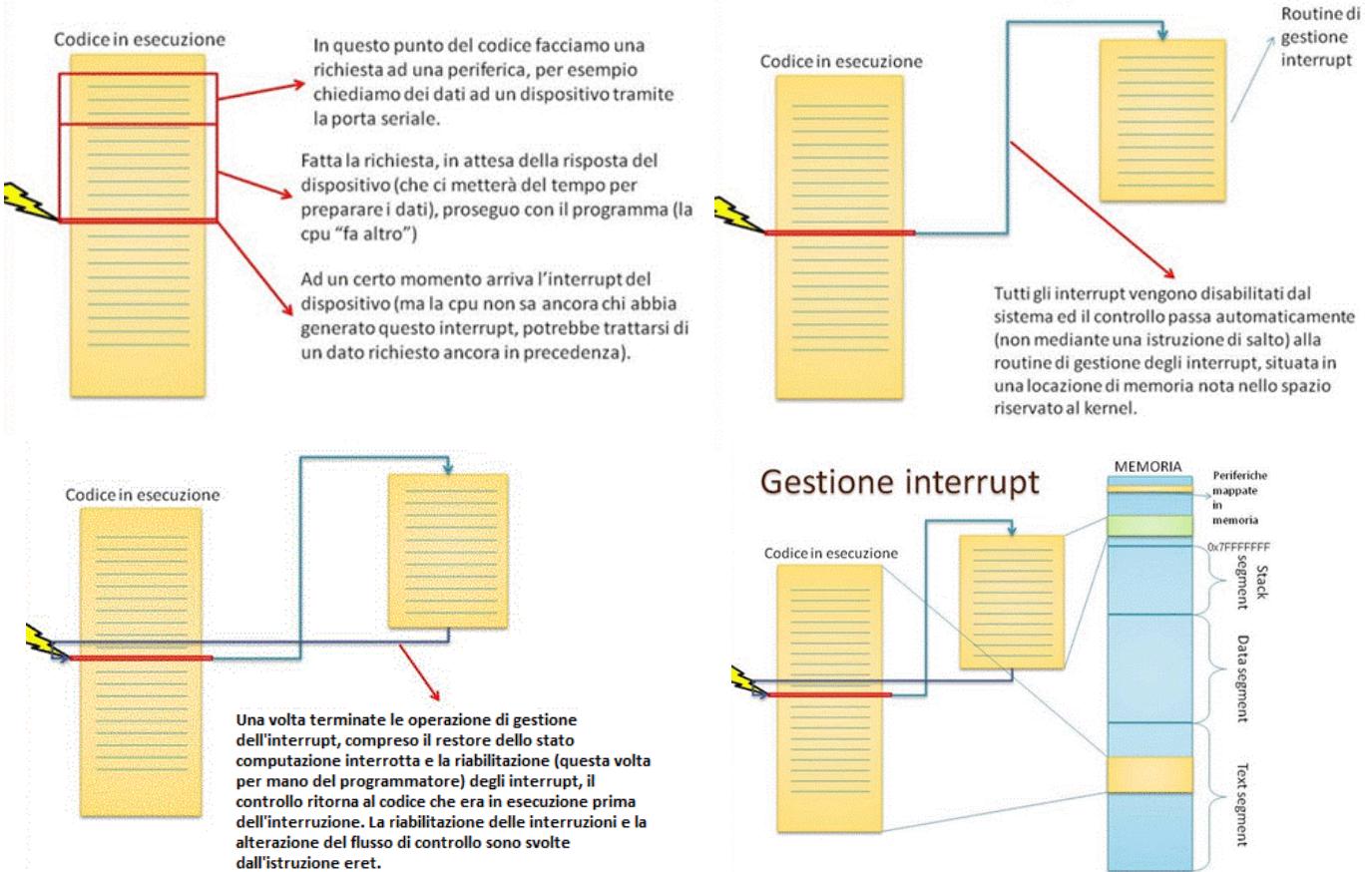
1) Controllo di Programma

- Tramite flusso di programma viene prelevato il bit ready dalla periferica mappata in memoria, confrontato nella CPU finché non risulta 1 (Busy Waiting)
- Quando il bit risulterà 1 viene eseguita una lw/sw sul registro dati della periferica
- LATENZA MINIMA → La CPU nota molto in fretta che la periferica è pronta (il tempo peggiore è il tempo di esecuzione di un ciclo di busy wait)
- ALTA BANDA PASSANTE → La CPU trasferisce subito il dato, di conseguenza la quantità di dati sarà molta in relazione al tempo

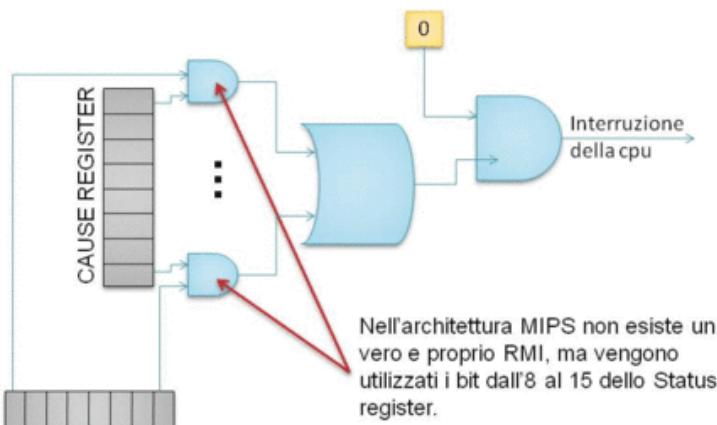


2) Gestione ad interrupt

- Linea di richiesta
- Quando una periferica genera un interrupt, la CPU deve eseguire una serie di istruzioni prestabilite, contenute a partire da una locazione di memoria nota a priori
 - SAVE
 - IDENTIFICAZIONE DELLA PERIFERICA INTERROMPENTE
 - GESTIONE DELLA PERIFERICA (TRASFERIMENTO DATO)
 - RESTORE
 - ERET (Ritorno dall'interrupt)



- Per identificare la periferica che ha generato l'interruzione, visto che la linea di richiesta è unica esistono delle strategie:
 - Si controllano le periferiche una ad una, capendo chi ha impostato il bit ready a 1
 - Vettorizzazione:
 - Base Vettore Interruzioni (Indirizzo di memoria noto) + Codice Periferica (ricevuto dalla linea di richiesta)
- Per non essere interrotti nuovamente durante la gestione di un interrupt esiste un meccanismo chiamato Mascheramento Globale (ogni bit mascheramento viene messo in and con i bit del cause reg)



- MINOR BANDA PASSANTE → Più istruzioni per identificare l'interrompente
- MAGGIORE LATENZA → La CPU si accorge dopo più tempo che la periferica ha mandato un segnale di interrupt

3) DMA (Direct Memory Access)

- Due registri aggiuntivi
 - L'indirizzo di memoria dal/al quale trasferire i dati
 - La quantità di dati da trasferire
- Fasi
 - Predisposizione registri
 - Stato
 - Dati
 - Puntatore alla memoria
 - Quantità dati da trasferire
 - Attivazione
 - La CPU mette a 1 un bit del registro stato della periferica
 - Da questo momento in poi la CPU può fare altro
 - Trasferimento elementare
 - Appena la periferica è pronta, accede alla memoria e trasferisce il contenuto del registro dati
 - Aggiorna il puntatore alla memoria e il contatore
 - Appena il contatore arriva a 0 viene generato un interrupt per notificare la CPU del trasferimento finito
- MASSIMA BANDA PASSANTE
- MINIMA LATENZA

Gerarchie di memoria

Introduzione

Un programma, in un certo istante di tempo, accede soltanto a una porzione relativamente piccola del suo spazio di indirizzamento

Due tipi di località:

- Temporale → Quando si fa riferimento ad un oggetto c'è la tendenza a fare riferimento allo stesso dopo poco tempo
- Spaziale → Quando si fa riferimento ad un elemento c'è la tendenza a fare riferimento ad elementi vicino allo stesso

Definizioni

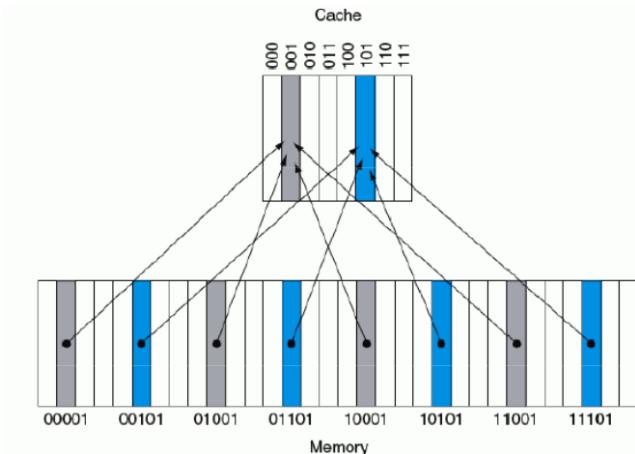
- Blocco → La più piccola quantità di informazione che può essere presente/assente in una gerarchia di memoria
- Hit → L'informazione richiesta dal processore si trova ai livelli più alti di gerarchia di memoria
- Miss → L'informazione richiesta dal processore NON si trova ai livelli più alti di gerarchia di memoria

Campi della cache

- Index
- Tag
- Valid Bit

Direct Mapped

A ciascun blocco della memoria corrisponde una specifica locazione nella cache



- Dimensione campo TAG $\rightarrow 32 - (2 + m + r)$
- Dimensione totale Cache $\rightarrow 2^n * (2^m * 32 + (32 - (n + m + 2)) + 1)$

$n = 2^n \rightarrow$ Blocchi della cache

$m = 2^m \rightarrow$ Word per ogni blocco della cache

Da quanti bit è costituita una cache a mappatura diretta contenente 16KB di dati e avente blocchi da 4 parole, ipotizzando un indirizzo di memoria di 32 bit?

- Capienza Cache $\Rightarrow 16 \text{ KB} \Rightarrow 16000 \text{ byte} \Rightarrow 16000 / 4 \Rightarrow 4000 \text{ words} \Rightarrow \text{ca. } 2^{12} \text{ words totali}$
- Numero Blocchi $\Rightarrow 4 \text{ words/blocco} \Rightarrow 4000 \text{ words} / 4 \Rightarrow 1000 \text{ words} \Rightarrow 2^{10} \text{ blocchi} \rightarrow m = 10$
- Word Blocco $\Rightarrow 4 \text{ words/blocco} \Rightarrow 2^2 \text{ words/blocco} \rightarrow n = 2$
- Bit/blocco $\Rightarrow 4 \text{ words} * 32 \text{ bit} \Rightarrow 128 \text{ bit/blocco}$
- CAPIENZA CACHE $= 2^n * (2^m * 32 + (32 - (n + m + 2)) + 1) \rightarrow 2^{10} * (2^2 * 32 + (32 - (2 + 10 + 2) + 1) = 2^{10} * 147 = 147 \text{ Kbit}$

Osservazioni

- Una cache con blocchi di dimensioni maggiori sfrutta maggiormente la località spaziale diminuendo la frequenza di miss
- La frequenza di miss torna a salire se la dimensione dei blocchi diventa troppo grande rispetto alla dimensione della cache \rightarrow Minor numero di blocchi, maggior competizione per occuparli \rightarrow La cache si ritroverà ad essere sovrascritta più frequentemente, causando quindi miss più frequenti \rightarrow Cresce inoltre il costo di una miss