REGISTRI livello piu' alto della gerarchia di memoria, dove processore memorizza dati.
**REGISTRI UTENTE**: usati da programmatore (in assembler o linguaggio alto livello) per memorizzare all'interno della CPU dati da elaborare. Visibili all'utente, si possono referenziare in istruzioni. Di tipo:
- uso generale (veramente generici -piu' flessibilita' e opzioni disponibili a basso livello ma istruzioni piu' lunghe-, o dedicati (es. accumulatore)); in genere tra 8 e 32 registri generali (in architetture RISC in genere più di 32)
- per dati
- per indirizzi (es. puntatori a segmento, indici, puntatori a pila)
- per codici condizione (es. risultato nullo, overflow, ecc. in genere istruzioni leggono implicitamente e non possono essere alterati da programmatore)
Quanto lunghi? Abbastanza per contenere indirizzo memoria, e abbastanza per una full word (unita' di riferimento per rappresentazione dati (alcune macchine consentono di combinare due registri per contenere valori di lunghezza doppia (es. double int a in C)
**REGISTRI DI CONTROLLO E DI STATO**: usati da unita' di controllo per monitorare operazioni processore e da programmi so per esecuzione programmi; non visibili all'utente. Tra questi:
- PC prossima istruzione
- IR istruzione corrente
- MAR contiene indirizzo di una locazione di memoria
- MBR contiene dato da (o per) memoria
- PSW (uno o piu' registri) mantiene informazioni di stato; tra i campi include: segno ultimo risultato, zero, riporto, uguale (op confronto), overflow, abilitazione/disabilitazione interrupt, supervisore (1 se in esecuzione programma di so)

**CICLO ESECUTIVO**
Dipende da architettura CPU, in generale:
- Instruction Fetch:
PC contiene indirizzo prossima istruzione -> indirizzo spostato in MAR -> posto su bus indirizzi -> unita' controllo richiede lettura in memoria -> risultato inviato nel bus dati -> copiato in MBR → copiato in IR -> PC incrementato
- Data Fetch:
IR esaminato -> se opcode indica indirizzamento indiretto: N bit piu' a destra di MBR vengono trasferiti in MAR, unita' di controllo richiede lettura in memoria, risultato posto in MBR, e solo allora avviene lettura operando (quindi serve connessione tra MBR e MAR)
- Execute:
varie forme, dipende da istruzioni; puo' includere: lettura/scrittura memoria, input/output, trasferimento dati tra registri, operazioni ALU
- Interrupt:
si salva contesto -contenuto PC deve essere salvato per permettere ripristino- PC copiato in MBR per essere scritto in memoria, indirizzo locazione di memoria speciale (es. stack pointer) copiato in MAR -> PC caricato con indirizzo prima istruzione routine di gestione dell'interrupt -> fetch istruzione puntata da PC
**PREFETCH**: fase fetch accede a memoria, fase execute di solito no, quindi si puo' prelevare istruzione successiva durante esecuzione istruzione corrente: idea di sfruttare tempo morto.
Non raddoppia prestazioni (es. jump puo' rendere vano prefetch)
Fase prelievo tipicamente piu' breve fase esecuzione, quindi soluzione ad esempio prefetch di piu' istruzioni o aggiungere piu' fasi per migliorare prestazioni (pipeline)

**LINGUAGGIO MACCHINA:**
insieme delle istruzioni che la CPU puo' eseguire
Elementi di un'istruzione macchina:
- codice operativo (solitamente nei bit piu' significativi) specifica operazione
- riferimento operando sorgente (facoltativo) uno o piu', input operazione
- riferimento operando risultato (facoltativo) destinazione
- riferimento istruzione successiva (implicito per maggior parte volte) es. per salti, chiamate
RAPPRESENTAZIONE ISTRUZIONE: divisa in campi, spesso usata rappresentazione simbolica (es. ABB SUB) per operazione e per operandi
TIPI ISTRUZIONE: Elaborazione dati, aritmetiche/logiche (di solito su registri), trasferimento dati tra registri o con memoria/dispositivi i/o, controllo del flusso del programma (salto)
Formato: in che modo vengono interpretati bit
Numero indirizzi: (zero, con indirizzi impliciti, o se utilizzo pila). Meno indirizzi -> istruzioni piu' elementari e piu' corte, quindi CPU meno complessa; ma piu' istruzioni per stesso programma quindi tempo piu' lungo esecuzione. Piu' indirizzi -> istruzioni piu' complesse e lunghe.

**TIPI OPERANDI**
- indirizzi: interi senza segno
- numeri (limite al modulo -in base a numero bit- e alla precisione -per numeri virgola mobile); spesso per op i/o si usano decimali impaccati -raggruppati per 4 bit- inefficiente, solo 10 di 16 configurazioni usate
- caratteri: codice ASCII (American Standard Code for Information Exchange); carattere = 7 bit -> 128 caratteri totali (alfabetici e di controllo), di solito ottavo bit per controllo parità o impostato a 0
- dati logici: bit considerati singolarmente
**TIPI OPERAZIONE**
- trasferimento dati, vanno specificati sorgente, destinazione, lunghezza dati da trasferire (e modo indirizzamento se necessario)
- aritmetiche, includono anche incremento, decremento, negazione
- logiche, operazioni sui bit (per manipolarli, anche in parallelo su tutti bit di un registro) es. shift e rotazione. Shift logico: bit vengono traslati, a una estremita' bit viene perso, all'altra introdotto 0; usato per isolare campi all'interno di una parola; Shift aritmetico: tratta dati come interi con segno che non viene traslato; Rotazione, shift ciclico, vengono mantenuti tutti i bit

- conversione
- i/o
- sistema, generalmente riservate a so
- **trasferimento controllo**, fa variare sequenza istruzioni, utile per eseguire piu' volte istruzione e per decidere in base a condizioni. Piu' comuni:
 . salto incondizionato: jump, skip (scavalca un'istruzione)
 . salto condizionato: tra operandi indirizzo a cui saltare se si verifica condizione, altrimenti istruzione successiva
 . **chiamata di procedura/subroutine**: programma autonomo all'interno del programma.
Procedura permette di usare piu' volte la stessa porzione di codice; consente modularità, suddividere grandi compiti in unita' piu' piccole; consente risparmio spazio, codice memorizzato una sola volta.
Prevede istruzioni di: chiamata CALL per saltare da locazione attuale a procedura, e di ritorno RETURN che rimanda all'istruzione che segue la chiamata. A ogni CALL corrisponde RETURN.
Procedura puo' essere chiamata da piu' indirizzi, e puo' essere all'interno di un'altra procedura. Processore deve salvare indirizzo di ritorno perche' questo avvenga correttamente. Usualmente: un registro, l'inizio della procedura chiamata, cima della pila.
Es chiamata CALL X (con RN registro specializzato e D lunghezza indirizzo) provoca:
_se approccio registro, RN<- PC+D e PC= X; procedura salva contenuto RN per successivo ritorno
_se memorizzare indirizzo ritorno a inizio procedura chiamata, X<- PC+D e PC<- X+1.
Entrambe non consentono pero' annidamento.
_se memorizzazione cima della pila, al momento chiamata posiziona indirizzo ritorno sulla pila (porzione di M dove letture/scritture avvengono sempre dalla cima); indirizzi memorizzati in cima uno dopo l'altro e ripresi sempre dalla cima, quindi nell'ordine inverso rispetto all'entrata -LIFO. Top della pila è indicizzato da apposito registro SP.

Per progettare insieme istruzioni serve considerare:
- repertorio: quante e quali operazioni
- tipi di dato
- formato: lunghezza, numero indirizzi, dimensione campi, ecc
- registri, quanti

**MODI INDIRIZZAMENTO**
- IMMEDIATO: operando parte dell'istruzione (campo indirizzo). Nessun accesso M, ma valore limitato alla dimensione del campo indirizzo
- DIRETTO: campo indirizzo=indirizzo operando. Singolo accesso in M, spazio indirizzamento limitato
- INDIRETTO: campo indirizzo=indirizzo cella M, che contiene indirizzo operando. Parole di lunghezza N permettono indirizzamento $2^N$ entità (in realta' campo indirizzo K, indirizzi effettivi referenziabili $2^K$) ma due accessi a M per un operando (utile per riferire intera memoria; indirettezza si puo' considerare come sottociclo del ciclo fetch/execute: istruzione prelevata viene esaminata per determinare se indirizzamento indiretto)
- REGISTRO: campo indirizzo=registro che contiene operando. Numero limitato registri, ma pochi bit necessari -> istruzioni più corte e fetch piu' veloce; nessun accesso M (efficiente per es per risultati intermedi calcoli, senza salvare dato in memoria)
- REGISTRO INDIRETTO: come per indirizzamento indiretto, registro punta a cella M che contiene operando. Un accesso in meno in M rispetto indirizzamento indiretto
- PILA: sequenza lineare di locazioni riservate in M.
Puntatore (registro SP stack pointer) con indirizzo cima della pila
Operando è su cima pila (quindi e' un esempio di indirizzamento registro indiretto)
- **SPIAZZAMENTO**: combina indirizzamenti diretto e registro indiretto. Campo indirizzo ha 2 sottocampi:
A= valore base (diretto);
R= registro il cui contenuto va sommato ad A per ottenere indirizzo operando (o viceversa, R base e A spiazzamento). Tre tipi comuni:
 . RELATIVO: R= registro PC-implicito; indirizzo operando= indirizzo istruzione corrente (PC) + campo indirizzo (trattato come numero in complemento a due – e' uno spiazzamento relativo rispetto l'istruzione corrente). Sfrutta concetto localita'.
 . REGISTRO-BASE: A contiene spiazzamento (solitamente intero senza segno) e R puntatore a indirizzo base (R implicito o esplicito)
 . INDICIZZAZIONE: (opposto rispetto registro-base) A contiene indirizzo memoria centrale e R contiene spiazzamento positivo da tale indirizzo (detto registro indice). Utile ad esempio per accedere a indirizzi memorizzati in sequenza: locazione partenza contenuto in campo indirizzo, registro indice parte da 0 e viene incrementato +1 dopo ogni accesso

**LUNGHEZZA ISTRUZIONI** condiziona, ed e' condizionata da:
- dimensione e organizzazione memoria
- struttura bus
- complessita' CPU, velocita' CPU
Compromesso tra repertorio di istruzioni potente e necessità di risparmiare spazio. Programmatori vogliono: più codici operativi e più operandi (per rendere programmi piu' brevi), più modi di indirizzamento (maggior flessibilità nell'implementare funzioni), e un intervallo di indirizzamento più ampio. Tutto ciò richiede bit e quindi istruzioni piu' lunghe. Altre considerazioni: lunghezza istruzioni dovrebbe essere uguale o multipla della larghezza del bus. Con istruzioni troppo lunghe, rischio memoria divenga collo di bottiglia se processore è in grado di eseguire istruzioni più velocemente del prelievo. Soluzioni: uso cache o istruzioni più brevi. Di conseguenza, istruzioni brevi prelevate a un tasso doppio, ma probabilmente non eseguite a una velocità doppia.
Altra caratteristica: lunghezza istruzioni dovrebbe essere multiplo lunghezza di un carattere (solitamente 8 bit) e numero virgola fissa. Dimensione parola uguale o correlata a dimensione trasferimento altrimenti spreco bit.

**FORMATO ISTRUZIONI** definisce disposizione bit (deve includere opcode e zero o piu' operandi (espliciti o no))
La maggior parte dei linguaggi macchina prevede più di un formato.

**ALLOCAZIONE DEI BIT** per data lunghezza, compromesso tra numero codici operativi e capacita' indirizzamento (piu' codici operativi richiedono piu' bit, meno disponibili per indirizzamento per data lunghezza).
Uso dei bit di indirizzamento e' determinato da:
- numero modi indirizzamento, se indicati esplicitamente (possono essere indicati implicitamente da opcode)
- numero operandi (di solito 1-2)
- registri/memoria: piu' registri si possono usare per riferimenti operandi, meno bit sono richiesti (di solito almeno 32)
- numero banchi registri (register file): vantaggio avere piu' di un banco e' che, a parita' di numero registri, una suddivisione funzionale utilizza meno bit nelle istruzioni: es. con 2 banchi da 8 registri, servono solo 3 bit per indicare registro (codice operativo identifica implicitamente quale banco)
- intervallo indirizzi: per indirizzi che referenziano la memoria e' correlato a numero di bit dell'indirizzo (per questo indirizzamento diretto usato raramente)
- granularita' indirizzi (es. byte o parola): es. indirizzamento al byte utile per manipolare caratteri ma richiede maggior numero bit
Esempi allocazione bit su macchine storiche:
Lunghezza FISSA:
**PDP-8**
Uno dei piu' semplici, molto vecchia: istruzioni lunghezza fissa 12 bit con formato variabile, opera su parole da 12 bit. Un solo registro per dati temporanei AC. Codice operativo di 3 bit (per allargare gruppo di operazioni, codice operativo 7 puo' riferire una microistruzione). I segnali di controllo sono contenuti in indirizzo.
**PDP-10**
Lunghezza e formato fissi, 36 bit. Introdotto concetto ortogonalita' (indipendentemente da cosa specifica opcode, gli altri campi sono tutti presenti -> opcode non dice come interpretare i bit; PDP-8 in base a opcode sappiamo come interpretare successivi bit). Stessi tipi operazioni per ogni tipo dato. Facilitava lavoro di programmatori e compilatori ma non utilizzava in modo efficiente spazio a disposizione.
Lunghezza VARIABILE: indirizzamento piu' flessibile ma aumento complessita' processore
(Formato ibrido: lunghezza variabile ma con numero finito varieta': es. in base a opcode tot operandi)
**PDP-11**
Gruppi di istruzioni a lunghezza diversa, all'interno dei quali ci sono formati diversi (esempio formato ibrido). Linguaggio potente e flessibile: linguaggi compatti ma aumenta costo hardware e complessita' programmazione.
**VAX**
Formato fortemente variabile (da 1 a 37 byte). L'idea era di fornire al programmatore istruzioni molto potenti (match 1:1 con linguaggio alto livello) scaricando pero' tutta complessita' su architettura. Gestione complessa per istruzioni che rallentava anche istruzioni semplici/frequenti, per avere a disposizione istruzioni potenti che venivano usate raramente.

**ARITMETICA CALCOLATORE** ogni componente ha scopo di portare dati in ALU. ALU gestisce numeri interi e reali (si e' consolidato uso due ALU specializzate).
Input ALU: segnali di controllo, dati da registri
Output ALU: flag (informazioni operazioni appena terminate), dato per registri
RAPPRESENTAZIONE INTERI
**IN MODULO E SEGNO**
Segno: bit piu' significativo (a sinistra: 0 positivo, 1 negativo)
Problemi: per eseguire op aritmetiche bisogna considerare sia moduli che segni; due rappresentazioni per lo 0 (diventa difficoltoso controllarlo); per questo rappresentazione poco utilizzata.
**IN COMPLEMENTO A DUE**
Segno come prima, differisce per l'interpretazione degli altri bit.
Nel complemento a due, si da un peso negativo al bit piu' significativo ($-2^{n-1}$).
Con sequenza di n bit $a_{n-1}\ a_{n-2}\ \dots\ a_1\ a_0$ ,
**numero A= $-2^{n-1} * a_{n-1} + \sum_{(i=0,\dots,n-2)} 2^i * a_i$**
che, se A e' positivo, il bit di segno $a_{n-1}$ e' 0 (e i restanti bit rappresentano il modulo)
se A e' negativo, il bit di segno $a_{n-1}$ e' 1
Questo e' quello che permette di evitare doppia rappresentazione dello zero.
(Quindi di fatto il bit piu' significativo rappresenta comunque il segno perche' per rappresentare numero negativo bisogna attivare bit piu' a sinistra)
- Numeri rappresentabili da $-2^{n-1}$ a $+2^{n-1}-1$ :
 .positivi: da 0 (n zeri) a $2^{n-1}-1$ (0 seguito da n-1 uni)
 .negativi: da -1 a $-2^{n-1}$ (bit segno 1)
(nota: es con 8 bit, $a_{n-1}$ =-128 (perche' diamo peso negativo) e i bit successivi con peso positivo andranno sommati e ne diminuiranno quindi il valore: 10000001 significa -128 (+1) quindi -127)
*PER RAPPRESENTAZIONE NEGATIVI:*
*dato k positivo, -k si ottiene complementando bit a bit e +1 al risultato :*
*5= 0101*
*-5= complemento 1010, e sommo 1 -> 1010+1= 1011*
*CASI SPECIALI:*
*- 0 e' invariante (0= 0000, complemento= 1111+1, 1 0000 con 1 a sx che e' overflow ed e' ignorato)*
*- $-2^{n-1}$ non puo' essere complementato (-8= 1000, complemento= 0111+1= 1000, quindi -(-8)=-8 !)*
*es. DA COMPLEMENTO A 2 A BASE 10: 11100(sappiamo che e' negativo)= 00011+1= 00100= -4*
NUMERI RAPPRESENTABILI:
complemento a 2 su 8 bit:
- numero piu' grande= $2^{n-1}-1= 2^{8-1}-1= +127$
- numero piu' piccolo= $-2^{n-1}= -2^{8-1}= -128$
CONVERSIONE TRA LUNGHEZZE DIVERSE:
- per modulo e segno: sposto bit segno nella posizione piu' a sinistra e riempio con tutti 0
- per complemento a due: sposto bit segno nella posizione piu' a sinistra e riempio replicando bit segno
**SOMMA E SOTTRAZIONE**
- SOMMA: normale somma binaria (controllo bit di segno per l'overflow)

- SOTTRAZIONE: necessari circuiti per somma e complemento. Minuendo – sottraendo: si considera opposto del sottraendo e lo si somma al minuendo (es. 7-5= 7+(-5)= 0111+1011= 0010)
HARDWARE: elemento centrale e' sommatore binario, al quale vengono forniti addendi e produce una somma e un segnale di overflow. Sommatore tratta numeri come interi senza segno. (i due numeri provengono da due registri A e B). Necessari segnali di controllo (uno switch) per sapere se usare valore registro B o il suo complemento (nel caso della sottrazione) da inviare al sommatore.
OVERFLOW: quando risultato somma supera il massimo (o minimo) valore rappresentabile con bit fissati. Si riconosce dal bit di segno: se 0(1) e i numeri sono entrambi negativi (positivi) -> overflow
(es. -4(1100)-1(1111)= 1 1011(-5): riporto, ma non overflow; +7(0111)+7(0111)= 1110 (non 14 ma -2): overflow)
**MOLTIPLICAZIONE E DIVISIONE**: piu' complesse
MOLTIPLICAZIONE INTERI SENZA SEGNO
1- implica il calcolo di prodotti parziali per ogni cifra del moltiplicatore (moltiplicando * moltiplicatore), che poi vengono sommati per generare prodotto finale.
2- per prodotti parziali: se bit moltiplicatore e' 0, il prodotto parziale e' 0; se bit moltiplicatore e' 1, il prodotto parziale e' il moltiplicando.
3- prodotto totale e' la somma dei parziali; per questa operazione, ogni prodotto parziale e' traslato di una posizione a sinistra rispetto a quello precedente.
4- due numeri binari di n bit producono un numero di 2n bit (es. 11*11=1001)
Per rendere piu' efficiente: i prodotti parziali possono essere sommati di volta in volta (elimina necessita' di memorizzarli); per risparmiare tempo sulla loro generazione, quando bit moltiplicatore e' 0 e' sufficiente la traslazione, senza la somma.
MOLTIPLICAZIONE CON NUMERI IN COMPLEMENTO A DUE
Per la somma, numeri in complemento a due vengono visti come interi senza segno. Per la moltiplicazione questo non funziona (se almeno uno dei due numeri e' negativo). Bisognerebbe usare rappresentazione complemento a due anche per prodotti parziali, ma avremmo comunque problemi nel caso di moltiplicatore negativo. Una possibile soluzione puo' essere quella di convertire fattori negativi in positivi, eseguire una normale moltiplicazione e, se necessario, cambiare segno del risultato. Una soluzione piu' rapida ed efficiente e' data dall'algoritmo di Booth.
DIVISIONE ancora piu' complessa della moltiplicazione (da implementare a livello hardware); basata sugli stessi principi generali: utilizza traslazioni, somme e sottrazioni ripetute.

**NUMERI REALI**
VIRGOLA FISSA
Assumendo un punto radice consente di rappresentare numeri frazionari. Numero fisso di bit per rappresentazione parte intera e decimale. Limiti: impossibilita' rappresentazione numeri molto grandi o frazioni molto piccole. Es. 1001.1010= $2^4+2^0+2^{-1}+2^{-3}$=9.625
**VIRGOLA MOBILE**
Bisogna specificare dove si trova la virgola.
Utilizza notazione scientifica. Consente di rappresentare numeri molto grandi e piccoli usando solo poche cifre.
Es. 976'000'000= $9{,}76*10^8$ rappresentato nella forma $\mathbf{\pm S*B^{\pm E}}$
S= significando o mantissa
B=base implicita, si assume che la virgola dopo la cifra (1) piu' significativa della mantissa
RAPPRESENTAZIONE tipica in 32 bit:
1 bit x segno - 8 bit x esponente polarizzato - 23 bit per mantissa
che rappresenta $\mathbf{\pm 1.S*2^E}$
**Esponente polarizzato** (biased representation): un valore fisso viene sottratto per ottenere il vero esponente:
- con k bit esponente binario, polarizzazione vale -> $2^{k-1}-1$
- **e= ep – ($2^{k-1}$-1)** quindi ep piu' piccolo sara' 0..0 -> facile confronto tra esponenti
NORMALIZZAZIONE
esponente e' aggiustato in modo che bit piu' significativo mantissa (implicito) sia 1 (es. $0{,}1*2^0= 1{,}0*2^{-1}$)
Es. 8 bit -> valori tra 0 e 255 → con polarizzazione $2^7$-1= 127 -> vero esponente da -127 a +128
PER ESERCIZI:
.esponente da decimale -> $+2^{k-1}$-1 , es. e=20 -> 20+127=147 -> 00010100+01111111= 10010011= 147
es. e=-20 -> -20+127=107 -> complemento 20= 11101011+1= 11101100+01111111= 01101011= 107
.esponente da binario -> $-2^{k-1}$-1, es. ep=01101011 -> 107-127=-20 -> 01101011-01111111= 11101100= -20
es. ep=10010011 -> 147-127=20 -> 10010011-01111111= 00010100= 20
.significando da binario -> 101...0= 1.101...0= $2^0+2^{-1}+2^{-3}$= 1+0,5+0,125= 1,625
.significando da decimale -> 14,3125 : calcolo separato per parte intera e frazionaria

```
  parte intera:   14|2  0 (resto)
                   7|2  1
                   3|2  1
                   1|2  1
                   0|
                          lettura resti dal basso: 1110=14

  parte decimale:  0,3125 *2 = 0 ,625
                   0,625  *2 = 1 ,25  -1= 0,25
                   0,25   *2 = 0 ,50
                   0,50   *2 = 1      -1= 0
                            lettura parti intere dei prodotti dall'alto: 0101
```

quindi: 1110.0101
normalizzato: $1.1100101*2^3$ -> ep= 3+127=130 -> 0000011+01111111= 10000010
in binario: 0(segno) 10000010(ep) 1100101..0(significando)

Es. $1.1010001*2^{10100}$= 0 10010011 10100010000000000000000= $1.6328125*2^{20}$
NUMERI RAPPRESENTABILI
- complemento a due: da $-2^{31}$ a $+2^{31}$-1

- virgola mobile:
 .esponente: da -127(tutti zeri) a 128(tutti uni)
 .significando: da 1.0..0(tutti zeri) a 1.1..1= $2-2^{-23}$ (23 bit significando)
   .negativi: da $-2^{128}*(2-2^{-23})$ a $-2^{-127}$
   .positivi: da $2^{-127}$ a $2^{128}*(2-2^{-23})$
NUMERI NON RAPPRESENTABILI
- negativi minori di da $-2^{128}*(2-2^{-23})$ [overflow negativo]
- negativi maggiori di da $-2^{-127}$ [underflow negativo]
- positivi minori di $2^{-127}$ [underflow positivo]
- positivi maggiori di $2^{128}*(2-2^{-23})$ [overflow positivo]
- non c'e' rappresentazione per lo 0
.underflow e' un problema meno critico poiche' numeri molto piccoli possono essere approssimati con lo 0
.non sono rappresentati piu' di $2^{32}$ numeri, ma sono divisi tra positivi e negativi
.numeri rappresentati non sono equidistanti tra loro: piu' densi vicino allo 0 (causa esponente) -> Questo e' il compromesso dell'aritmetica in virgola mobile: molti calcoli producono risultati che non sono esatti e devono essere arrotondati al valore più vicino che la notazione può rappresentare.
PRECISIONE E DENSITA' se si aumentano bit esponente, cresce intervallo rappresentabile ma diminuisce densita'. Per avere piu' precisione l'unica alternativa e' aumentare numero bit -> precisione singola (32 bit) e precisione doppia 64 bit

**STANDARD IEEE 754**

Standard per rappresentazione numeri in virgola mobile, sviluppato per facilitare portabilita' programmi. Usato su praticamente tutti processori moderni.
- Definisce formato singolo a 32 bit e formato doppio a 64 bit.
- Formato singolo, esponente 8 bit. Formato doppio, esponente 11 bit.
- 1 implicito a sinistra virgola, base 2
- Due formati estesi singolo e doppio: piu' bit per esponente e significando, usati per calcoli intermedi. Precisione superiore -> diminuiscono possibilita' risultato finale con eccessivo arrotondamento e -con intervallo piu' ampio- di overflow intermedio che causerebbe terminazione calcolo -> formato singolo esteso racchiude benefici formato doppio senza incorrere in penalita' tempo associata a precisione piu' alta.
- Esponente polarizzato da 1 a 254 (cioe' da -126 a +127) – questo perche' valori estremi hanno valori speciali:
 .esponente 0, mantissa 0: rappresenta 0 positivo e negativo
 .esponente 1..1, mantissa 0: infinito positivo e negativo [errore overflow o valore infinito come risultato]
 .esponente 0, mantissa non nulla: numero denormalizzato. In questo caso il bit a sinistra del punto binario e' 0 e il vero esponente e' -126
 .esponente 1..1, mantissa non nulla: errore (NAN), usato per segnalare condizioni di errore

**ARITMETICA IN VIRGOLA MOBILE**

Occorre allineare operandi aggiustando esponenti (per somma/sottrazione) -per avere stessi esponenti

**SOMMA E SOTTRAZIONE**

.Necessario assicurarsi che operandi abbiamo stesso esponente -> puo' essere richiesta traslazione punto radice in uno degli operandi per ottenere l'allineamento.
.Operandi devono essere trasferiti nei registri di input per ALU, se bit implicito per il significando esso deve essere reso esplicito.
Quattro fasi:
(0- Processo inizia cambiando il segno del sottraendo, nel caso di sottrazione)
1- controllo dello 0 (non occorre effettuare operazione, il risultato e' l'operando non nullo)
2- allineamento significandi (rendere uguali esponenti), manipolazione addendi. Lo standard e' aumentare l'esponente piu' piccolo. Si ottiene l'allineamento traslando ripetutamente la parte del modulo del significando a destra di una cifra e incrementando l'esponente.
3- somma dei significandi, tenendo conto del loro segno. Eventuale overflow del significando nel risultato viene riportato e comporta la sospensione dell'operazione.
4- normalizzazione del risultato (traslazione a sinistra finche' cifra piu' significativa e' 1). Ogni traslazione provoca un decremento dell'esponente e pertanto causa potenzialmente un underflow di esponente. Infine, il risultato deve essere arrotondato.
Es. in base 10: $(123*10^0)+(456*10^{-2})$ -> allineiamo esponenti: $(123*10^0)+(4,56*10^0)$ -> sommo mantisse: 123+4,56= 127,56 → risultato: $127,56*10^0$

**MOLTIPLICAZIONE E DIVISIONE**

1- controllo dello 0
2- somma esponenti
3- sottrazione polarizzazione (perche' sommando due esponenti, sommo due polarizzazioni); il risultato potrebbe comportare overflow o underflow dell'esponente, che causerebbe conclusione dell'algoritmo.
4- moltiplicazione/divisione operandi, tenendo conto dei segni
5- arrotondamento
(in caso di DIVISIONE: sottrazione degli esponenti, la polarizzazione va aggiunta)
BIT DI GUARDIA per precisione risultato
Prima di operazione in virgola mobile, esponente e significando degli operandi vengono caricati in registri ALU. Lunghezza registro e' generalmente superiore a quella del significando + 1 bit. Nel registro, i bit più a destra sono messi a 0 e permettono di non perdere bit se i numeri vengono shiftati a destra.
ARROTONDAMENTO
Se il risultato di un'operazione e' memorizzato in un registro più lungo, quando lo si riporta nel formato in virgola mobile, bisogna arrotondarlo. Standard IEEE 754 prevede 4 approcci:
- arrotondamento al piu' vicino, tra i numeri rappresentabili (default). Se bit aggiuntivi iniziano con 1: sommo 1; se bit aggiuntivi iniziano con 0: troncamento.
- arrotondamento a +infinito, per eccesso
- arrotondamento a -infinito, per difetto
- arrotondamento a 0 (cioe' troncamento dei bit in piu')

**PIPELINE**

Idea -> Parallelismo: se un lavoro non puo' essere svolto piu' velocemente da una sola unita', conviene decomporlo in parti che possano essere eseguite da piu' unita' contemporaneamente -> catena di montaggio.
Pipeline ad esecutori specializzati -> Ogni esecutore svolge sempre la stessa fase di lavoro -> Soluzione efficace in termini di uso di risorse (3T/N lavori con N/3 risorse)
DECOMPOSIZIONE IN FASI
Esecuzione di una generica istruzione puo' essere suddivisa nelle seguenti fasi:
- fetch (FI) dell'istruzione
- decodifica (DI) dell'istruzione
- calcolo indirizzo operando (CO)
- fetch operandi (FO) lettura degli operandi in memoria
- esecuzione (EI) dell'istruzione
- scrittura (WO) del risultato in memoria

Vari fenomeni pregiudicano il raggiungimento del massimo parallelismo teorico:

**1-SBILANCIAMENTO DELLE FASI**
Durata diversa per fase e per istruzione. La suddivisione in fasi va fatta in base all'istruzione piu' onerosa. Non tutte le istruzioni richiedono le stesse fasi e risorse, e non tutte le fasi richiedono lo stesso tempo di esecuzione (es. lettura operando tramite indirizzamento registro o indiretto)
Soluzioni:
.decomporre fasi piu' onerose in piu' sottofasi (costo elevato e bassa utilizzazione)
.duplicare gli esecutori delle fasi piu' onerose e farli operare in parallelo (es. ALU per aritmetica intera e ALU per aritmetica a virgola mobile)

**2-PROBLEMI STRUTTURALI**
Quando due o piu' istruzioni nella pipeline hanno bisogno della stessa risorsa causano conflitti di accesso (es. la memoria per gli stadi FI, FO, WO)
Soluzioni:
.suddividere le memorie (accessi paralleli: cache per istruzioni e cache per dati)
.introdurre fasi non operative (nop)

**3-DIPENDENZA DAI DATI**
Si ha quando l'operazione successiva dipende dal risultato dell'operazione precedente (dato modificato nella fase EI dell'istruzione corrente deve essere utilizzato nella fase FO dell'istruzione successiva)
Tre tipi di dipendenza:
RAW: Read After Write, istruzione 1 scrive, istruzione 2 legge, rischio lettura avvenga prima di scrittura
WAR: Write After Read
WAW: Write After Write
Soluzioni:
.introdurre fasi non operative (nop)
.risoluzione a livello di compilatore: esempio su pipeline MIPS: il compilatore fa analisi prima che programma vada in esecuzione, per capire se istruzioni che creano dipendenza possano essere spostate (distanziandole) all'interno della sequenza di istruzioni di un programma senza modificare la semantica del programma
.riordino delle istruzioni (pipeline scheduling). Stesso di sopra, ma fatto al momento dell'esecuzione
**.data forwarding**: individuazione del rischio e prelievo del dato all'uscita della ALU (se ne fa una copia). E' importante capire il momento in cui il dato viene creato. Il forward e' possibile grazie ad appositi circuiti di bypass e MUX, regolati da Unita' di controllo e altre unita', che permettono di mandare in input alla ALU dati derivanti dalla memoria o dati che la ALU stessa ha mandato in output nel ciclo precedente.
L'implementazione del data forwarding dipende dall'architettura. Es. nell'architettura MIPS vi è un apposito circuito di identificazione delle dipendenze e gestione del data forwarding.
Tale circuito ha tre componenti fondamentali:
.Forwarding Unit: unita' che decide se attivare il forward attivando nell'opportuno modo i mutex della ALU
.Hazard, detection Unit: unita' in grado di riconoscere le dipendenze e di generare stalli in caso di dipendenze non risolvibili
.Control Unit: manda segnali di controllo che regolano il forward. Nel caso in cui la dipendenza venga rilevata come risolvibile, allora nelle MIPS sara' possibile fare il forward di dati da fase EX a EX, e da fase MEM a EX.

**4-DIPENDENZA DAI CONTROLLI**
Quando istruzioni causano una violazione della sequenzialita'.
Istruzioni che modificano il PC (salti condizionati e non, chiamate/ritorni procedure, interruzioni) invalidano la pipeline -> La fase fetch successiva carica l'istruzione seguente, che può non essere quella giusta -> Nel caso di salti condizionati, si sa se il salto verra' preso solo dopo fase esecutiva.
Tali istruzioni sono circa il 30% del totale medio di un programma.
Soluzioni:
.mettere in stallo pipeline fino al calcolo dell'indirizzo prossima istruzione (semplice ma non efficiente)
.individuare istruzioni critiche e anticiparne esecuzione, eventualmente mediante apposita logica di controllo (compilazione complessa, hardware specifico)
Soluzioni per salti condizionati:
**.Flussi multipli** (multiple streams): replicare parti iniziali della pipeline, per prelevare entrambe le istruzioni facendo uso di due flussi, uno che contenga istruzione successiva (salto non preso), e l'altro istruzione destinazione (target) del salto (salto preso)
Problemi di questa soluzione:
-conflitti nell'accesso a risorse (registri, memoria, ALU,...) da parte delle 2 pipeline
-in ciascuno dei flussi si possono verificare altre istruzioni di salto, prima di aver risolto la decisione originaria. Ogni istruzione di tale tipo richiede un flusso aggiuntivo.
**.Prelievo anticipato destinazione** (prefetch branch target): si effettua fetch anticipato anche dell'istruzione di destinazione del salto in modo da trovarla gia' caricata nel caso in cui il salto debba avvenire.
Problemi di questa soluzione:
-non evita eventuale svuotamento della pipeline con conseguente perdita di prestazioni
**.Buffer circolare** (loop buffer): e' una memoria piccola e molto veloce dove mantenere le ultime n istruzioni prelevate. In caso di salto, si controlla se l'istruzione destinazione e' presente nel buffer, cosi' da evitare il fetch della stessa.

Tre vantaggi:
-se salto condizionale realizza un ciclo, le istruzioni possono essere gia' tutte contenute nel buffer e non c'e' bisogno di effettuare fetch di nuovo
-anticipando il fetch, buffer circolare conterra' alcune istruzioni successive alla corrente, che quindi in caso di non salto saranno gia' presenti senza caricarle dalla memoria memoria
-se si salta in avanti di poche istruzioni, 'istruzione destinazione sara' gia' nel buffer (tipico if-then-else)
.**Predizione del salto** (branch prediction): si cerca di prevedere se il salto sara' intrapreso oppure no.
Approcci statici:
-previsione di saltare sempre
-previsione di non saltare mai (piu' diffuso)
-previsione in base al codice operativo
Approcci dinamici:
(cercano di migliorare accuratezza previsione memorizzando storia delle istruzioni di salto condizionato)
-bit taken/not taken :ad ogni istruzione di salto condizionato si associano uno o più bit che codificano la storia recente -> spingono processore a prendere decisione alla successiva occorrenza dell'istruzione (generalmente bit non in memoria centrale ma in locazione temporanea ad alta velocita').
 .predizione con 1 bit: si memorizza esito salto ultima esecuzione; non ideale in cicli, dove salti sono presi di frequente: si verificheranno due errori di previsione ad ogni utilizzo del ciclo, all'ingresso e all'uscita.
 .predizione con 2 bit: possibile memorizzare risultato delle ultime due istanze di esecuzione dell'istruzione associata. Fino a quando le successive istruzioni di salto condizionato vengono effettuate, il processo decisionale prevede che salto sara' intrapreso. Se una singola previsione e' errata, l'algoritmo continua a prevedere che il prossimo salto sara' effettuato. L'algoritmo richiede due previsioni errate consecutive prima di cambiare la propria decisione di predizione.
Problemi di questa soluzione:
-quando si decide di saltare, bisogna aspettare la decodifica dell'indirizzo destinazione prima di poter prelevare l'istruzione. Si puo' anticipare il prelievo a patto di salvare opportune info nella branch history table (tabella storia dei salti, piccola memoria associata allo stadio fetch della pipeline):
Ogni riga della tabella e' costituita da 3 elementi:
-indirizzo istruzione salto
-indirizzo destinazione del salto (o l'istruzione destinazione stessa)
-alcuni bit di storia che descrivono lo stato dell'uso dell'istruzione
Ogni prefetch innesca una ricerca nella tabella: se c'e' una corrispondenza, si effettua previsione sulla base dello stato dell'istruzione che, nel caso in cui risulti errato, viene aggiornato. Se istruzione salto non e' presente in tabella, viene aggiunta cancellando una delle righe presenti.
.**Salto ritardato** (delayed branch): l'idea e' quella di utilizzare gli stadi inattivi a causa dello stallo per fare del lavoro utile (in attesa di verificare la condizione di salto)
-la CPU esegue sempre l'istruzione che segue il salto e solo dopo altera, se necessario, la sequenza di esecuzione delle istruzioni
-l'istruzione che segue quella di salto si dice essere posta nel branch delay slot (che deve essere indipendente dall'esito del salto)
-il compilatore cerca di allocare nel branch delay slot una istruzione opportuna
Di fatto quindi il salto non viene ritardato ma l'intervallo precedente a esso viene usato per eseguire una qualche istruzione che sia indipendente dal salto stesso.

**CISC**
Complex Instruction Set Computer
Architettura nata con lo scopo di migliorare l'efficienza dell'esecuzione dei programmi, facilitare il lavoro del compilatore e supportare linguaggi ad alto livello piu' complessi, per ridurre gap semantico tra operazioni macchina e linguaggi alto livello.
Implementa operazioni complesse tramite microcodice, idea di incremento prestazionale non dovendo fare una successione di istruzioni primitive per eseguire l'istruzione complessa.
Scopo di ottenere programmi piu' piccoli/compatti, eseguiti più velocemente
(Prevalgono le implementazioni in hardware poiche' meno costoso del software), e permette grande flessibilita' grazie a set istruzioni molto ampio e svariati modi di indirizzamento.
Dire che architetture CISC semplifichino il compito del compilatore e' controverso, le istruzioni macchina sono complesse difficili da sfruttare, quindi l'ottimizzazione e' piu' difficile.
Inoltre i programmi occupano meno memoria, ma la memoria e' diventata economica. Possono non occupare meno bit, ma semplicemente sembrano piu' corti in forma simbolica (codice mnemonico).
Le istruzioni piu' semplici (quelle piu' usate) diventano piu' lente a causa della complessita'
**RISC**
Reduced Instruction Set Computer, caratteristiche:
- numero elevato di registri ad uso generale
- oppure uso compilatori per ottimizzare uso dei registri
(cio' suggerisce prestazioni migliori per riduzione accessi memoria e aumento riferimenti a registri)
- set istruzioni semplice e limitato
- ottimizzazione della pipeline (basata su formato fisso istruzioni, modi indirizzamento semplici, ecc.)
- controllo cablato (hardware, meno flessibile ma piu' veloce) e non microprogrammato

Da studio per stabilire quale filosofia risulta migliore non emerge vincitore netto (molti processori utilizzano idee di entrambe le filosofie). Ha pero' evidenziato necessita' ricerca di un rapido accesso agli operandi, per cui la memorizzazione in registri risulta privilegiata; banco registri e' alloggiato nello stesso chip di ALU e unita' di controllo. Operandi usati di frequente devono essere mantenuti in registri e per fare cio': approccio software, affidarsi a compilatore; approccio hardware, usare piu' registri (necessaria organizzazione)

**TRATTAMENTO VARIABILI LOCALI**
**FINESTRE DI REGISTRI**
- Osservazioni su chiamate di procedure:

.non presentano grado di annidamento elevato
.tipicamente utilizzano pochi parametri e variabili scalari locali (problema: a ciascuna chiamata, variabili locali devono essere copiate in memoria, affinche' programma chiamato possa utilizzare i registri, e occorre effettuare passaggio parametri)
- Soluzione:
.utilizzare tanti registri suddividendoli in gruppi -> a ogni chiamata di procedura e' assegnato nuovo gruppo registri; finestre di registri adiacenti si sovrappongono per consentire il passaggio dei parametri.
Ogni gruppo registri e' suddiviso in tre sottogruppi:
-registri che contengono parametri passati alla procedura chiamata
-registri che memorizzano il contenuto delle variabili locali
-registri temporanei, per scambiare parametri e risultati con livello inferiore
Registri temporanei di un livello sono allineati con registri parametri del livello inferiore, cioe' sono fisicamente gli stessi registri -> permette passaggio parametri senza spostare dati
Quindi, quando avviene chiamata:
.puntatore alla finestra corrente (Current Window Pointer) viene aggiornato per mostrare finestra registri attiva
.se si esaurisce capacita' buffer (tutte finestre in uso a causa chiamate annidate) viene generata interruzione e finestra piu' datata viene salvata in memoria
.un puntatore (Saved Window Pointer) indica dove si deve ripristinare l'ultima finestra salvata in memoria.

**TRATTAMENTO VARIABILI GLOBALI**

Schema a finestre appena descritto tratta variabili scalari locali.
Necessari registri globali a disposizione di tutte le procedure.
Ottimizzazione dei registri tramite compilatore:
-si assume numero limitato registri (16-32)
-compilatore assegna un registro simbolico (o virtuale) ad ogni variabile candidata
-quindi stabilisce una corrispondenza tra registri simbolici (numero virtualmente illimitato) e registri reali
-registri simbolici il cui uso non si sovrappone possono condividere lo stesso registro reale
-se i registri reali non sono sufficienti per contenere tutte le variabili riferite in un dato intervallo di tempo, alcune variabili vengono mantenute in memoria principale
L'essenza dell'ottimizzazione e' decidere quali variabili assegnare ai registri in ogni dato punto del programma.
Tecnica piu' comune nei compilatori RISC e' una forma di **colorazione di un grafo**, vista nel seguente modo: dato un grafo, si vogliono assegnare dei colori (che rappresentano di registri reali) ai suoi nodi (che rappresentano registri simbolici) in modo che nodi adiacenti (cioe' registri simbolici attivi nello stesso momento) non abbiano lo stesso colore, e che il numero di colori usati sia minimo. Più registri simbolici possono quindi essere mappati su uno stesso registro reale se il loro uso non si sovrappone temporalmente. Nodi che, con questa tecnica, non possono essere colorati, devono essere posti in memoria.

CONFRONTO BANCO REGISTRI BASATO SU FINESTRE-CACHE
- Banco registri contiene variabili scalari locali delle N-1 procedure attivate. Cache contiene una selezione delle variabili scalari usate di recente.
- Banco dei registri e' piu' veloce, ma cache usa lo spazio in modo piu' efficiente, dato che opera dinamicamente.
- Banco dei registri fa un uso inefficiente dello spazio quando non tutte le procedure richiedono intera dimensione della finestra ad esse allocata. D'altra parte, cache importa un blocco di dati che potrebbe non essere completamente utilizzato.
- Cache e' in grado di trattare variabili globali e locali e scoprire dinamicamente quali variabili globali sono maggiormente utilizzate e contenerle in memoria. Invece per banco registri questo compito e' affidato al compilatore, compito difficile.
- In banco registri trasferimento dati tra registri e memoria e' determinato dalla profondita' di annidamento della procedura (solitamente poca) quindi uso memoria relativamente poco frequente.
- In banco registri per accedere a variabile locale si utilizza indirizzamento a registro, molto semplice e veloce. In cache invece indirizzamento e' molto più lento.

**famiglia MIPS** esempio architettura RISC
- architettura molto regolare con insieme di istruzioni semplice e compatto
- architettura progettata per una implementazione efficiente della pipeline
- istruzioni a lunghezza fissa 32 bit, con pochi formati
- co-processore per istruzioni a virgola mobile e gestione delle eccezioni
- vengono utilizzati meccanismi per il trattamento dei conflitti sui dati

REGISTRI
- 32 registri a 32 bit (registro 0 usato sempre per valore 0, non puo' essere scritto)
- architettura load/store (accesso alla memoria avviene solo tramite queste due istruzioni, le altre operano solo su registri):
 .istruzioni trasferimento per muovere dati tra memoria e registri
 .istruzioni per manipolazione dati operano su valori dei registri
 .nessuna operazione memoria-memoria
(- nelle istruzioni, il registro destinazione e' sempre esplicito)

DATI E MODI INDIRIZZAMENTO
- registri possono essere caricati con byte, mezze parole (16 bit), e parole (32 bit, 4 byte)
 (riempiendo con 0 oppure estendendo il segno sui bit non coinvolti del registro -per num in complemento a due)
- modalita' di indirizzamento ammesse:
 .immediata :  es.  addi $2, $2, 0004
 .displacement :  es.  sw $1, 000c($1)  <- leggo $1 e store in indirizzo 000c+contenuto $1
- modalita' di indirizzamento derivate:
 .indiretta registro  es.  sw $2, 0000($3)
 .assoluta  es.  lw $1, 00c4($0)

FORMATO ISTRUZIONI
- uso di formato fisso semplifica fetch e decodifica
- tre formati per le istruzioni, con stessi campi per codice operativo, e parzialmente per riferimenti a registri

- tutti i riferimenti a memoria consistono in spiazzamento a 16 bit relativo ad un registro
- 32 bit per tutti i formati

**Formato R (registro)**
32 bit divisi in: 6 opcode - 5 rs - 5 rt - 5 rd - 5 shamt - 6 funct
.rs: source, primo argomento
.rt: target, secondo argomento
.rd: destination
.shamt: parametri di shift
.funct: estensione opcode, per gestire tipi dati
Quindi riferimento a 3 registri, due per operandi (rs-rt), uno per destinazione (rd).
es. add $8, $5, $8 [R8] <- [R5]+[R8]
rd rs rt

**Formato I (Immediato)** per load/store, op aritmetico-logiche con dato immediato, salti condizionati
32 bit divisi in: 6 opcode - 5 rs - 5 rt - 16 address
.address: puo' essere usato come displacement o come dato immediato
es. lw $8, 1200($9) [R8] <- Mem[1200+[R9]]
rt rs
Quindi riferimento a due soli registri, uno operando (rs), e uno destinazione (rt)

**Formato J (Jump)** per salto incondizionato
32 bit divisi in: 6 opcode - 26 address
es. j 45054 [PC] <- 45054
Quindi quantita' indirizzabile piu' piccola della totale (26<32)

FASI MIPS (senza pipeline)
1- **IF** instruction fetch:
con NPC registro temporaneo
*IR <- Mem[PC];*
*NPC <- PC+4;*
2- **ID** instruction decode/register fetch cycle:
con A, B, Imm registro temporanei
*A <- Regs[rs];*
*B <- Regs[rt];*
*Imm <- campo immediato di IR con segno esteso;*
3- **EX** execution
-riferimento memoria: *es. lw $8,1200($9)*
*ALUOutput <- A + Imm;*
-istruzione ALU registro-registro: *es. add $8,$9,$8*
*ALUOutput <- A func B;*
-istruzione ALU registro-immediato: *es. addi $8,$9,4*
*ALUOutput <- A op Imm;*
-salto: *es. j 45054*
*ALUOutput <- NPC + (Imm<<2);*
*Cond <- (A==0);*
4- **MEM** memory access
*PC <- NPC; in tutti i casi*
-riferimento a memoria: *es. lw $8,1200($9)*
*LMD <- Mem[ALUOutput];*
or *es. sw $5,1700($4)*
*Mem[ALUOutput] <- B;*
-salto: *es. j 45054*
*if(Cond) PC <- ALUOutput;*
5- **WB** write/back
-istruzione ALU registro-registro: *es. add $8,$9,$8*
*Regs[rd] <- ALUOutput;*
-istruzione ALU registro-immediato: *es. addi $8,$9,4*
*Regs[rt] <- ALUOutput;*
-istruzione load: *es. lw $8,1200($9)*
*Regs[rt] <- LMD;*

**PIPELINE MIPS**
- architettura che si presta facilmente alla pipeline: uno stadio per fase, 1 ciclo di clock per stadio
- occorre memorizzare i dati fra una fase e successiva: introdotti registri (pipeline registers) fra i vari stadi
- tali registri memorizzano dati e segnali di controllo che devono transitare da uno stadio al successivo
- dati che servono a stadi non immediatamente successivi vengono comunque copiati nei registri dello stato successivo per garantire la correttezza dei dati
- quando istruzione passa da fase ID a EX si dice che e' stata rilasciata (issued)
- e' possibile individuare tutte le dipendenze dai dati nella fase ID
- se si rileva dipendenza, questa va in stallo prima di essere rilasciata
- inoltre, sempre nella fase ID, e' possibile determinare tipo di data forwarding da adottare per evitare stallo e predisporre gli opportuni segnali di controllo
- registri letti e scritti nello stesso ciclo di clock (evita conflitti tra ID e WB); ciclo clock diviso in 2: si eseguono scritture nella prima meta' del ciclo, e letture nella seconda meta' del ciclo

**RILEVAZIONE DIPENDENZA DEI DATI**
Da parte di stadio ID, grazie ad un apposito circuito di identificazione delle dipendenze e gestione del data forwarding. Tale circuito ha tre componenti fondamentali:
- Forwarding Unit: unita' che decide se attivare il forward attivando nell'opportuno modo i MUX della ALU
- Hazard detection Unit: unita' che riconosce dipendenze e genera stalli se risultano non risolvibili

- Control Unit: manda segnali di controllo che regolano forward, e dati che devono essere mandati (inoltre manda segnali di controllo che regolano esecuzione e utilizzo dell'hardware per memorizzazione ed write back)
Forwarding Unit rileva dipendenze confrontando, mediante comparatori:
.registri associati ai campi di input dell'istruzione che si trovano nello stadio ID
.con registri target delle istruzioni precedente non ancora terminate
.comparatori possono leggere registri associati ai campi di input e output facendo riferimento al campo IR dei banchi di registri.
.istruzione in stadio ID avra' informazioni su registri/o di input nel banco IF/ID, mentre registro istruzione con cui fare confronto sara' nel banco ID/EX o EX/MEM o MEM/WB.
.se i due registri sono uguali e non vi e' possibilita' di data forwarding viene generato uno stallo
In particolare:
-istruzione formato R, campi di input sono rs (IF/ID.IR[rs]) ed rt (IF/ID.IR[rt]) e verranno confrontati con: rd se istruzione precedente ha formato R, o rt se istruzione precedente ha formato I
-istruzione formato I, campo di input e' rs (IF/ID.IR[rs]) e verra' confrontato con:
rd se istruzione precedente ha formato R o rt se istruzione precedente ha formato I

| Stage | Any instruction | | |
|---|---|---|---|
| IF | IF/ID.IR ← Mem[PC]<br>IF/ID.NPC,PC ← (if ((EX/MEM.opcode == branch) && EX/MEM.cond){EX/MEM.ALUOutput}<br>else {PC+4}); | | |
| ID | ID/EX.A ← Regs[IF/ID.IR[rs]]; ID/EX.B ← Regs[IF/ID.IR[rt]];<br>ID/EX.NPC ← IF/ID.NPC; ID/EX.IR ← IF/ID.IR;<br>ID/EX.Imm ← sign-extend (IF/ID.IR[immediate field]); | | |
| | ALU instruction | Load or store instruction | Branch instruction |
| EX | EX/MEM.IR ← ID/EX.IR;<br>EX/MEM.ALUOutput ←<br>ID/EX.A *op* ID/EX.B;<br>or<br>EX/MEM.ALUOutput ←<br>ID/EX.A *op* ID/EX.Imm; | EX/MEM.IR ← ID/EX.IR;<br>EX/MEM.ALUOutput ←<br>ID/EX.A + ID/EX.Imm;<br><br>EX/MEM.B ← ID/EX.B | <br>EX/MEM.ALUOutput ←<br>ID/EX.NPC + (ID/EX.Imm << 2);<br><br>EX/MEM.cond ← (ID/EX.A == 0); |
| MEM | MEM/WB.IR ← EX/MEM.IR;<br>MEM/WB.ALUOputput ←<br>EX/MEM.ALUOutput; | MEM/WB.IR ← EX/MEM.IR;<br>MEM/WB.LMD ←<br>Mem[EX/MEM.ALUOutput];<br>or<br>Mem[EX/MEM.ALUOutput]<br>← EX/MEM.B; | |
| WB | Regs[MEM/WB.IR[rd]] ←<br>MEM/WB.ALUOutput;<br>or<br>Regs[MEM/WB.IR[rt]] ←<br>MEM/WB.ALUOutput; | For load only:<br><br>Regs[MEM/WB.IR[rt]] ←<br>MEM/WB.LMD; | |

**MICROPROGRAMMAZIONE**
La microprogrammazione e' utilizzata per implementare l'unita' di controllo della CPU la quale, grazie al microprogramma, implementa ogni istruzione tramite una sequenza di micro-operazioni eseguite direttamente dall'hardware e di generare nella giusta sequenza i segnali di controllo che provocano l'esecuzione di ogni operazione elementare. Il microprogramma (firmware) nell'unita' di controllo ha una struttura ciclica in cui alterna l'esecuzione di una operazione speciale con l'esecuzione di un'operazione esterna il cui codice e dati da elaborare sono stati acquisiti dalla operazione speciale. Il microprogramma riunisce quindi tutte le micro-operazioni necessarie per effettuare l'operazione speciale del microprogramma e le micro-operazioni necessarie per effettuare ogni operazione esterna. In generale la parte operativa invia all'unita' di controllo delle variabili di condizionamento. In base a queste variabili, l'unita' di controllo manda dei segnali di controllo che designano la micro-operazione da eseguire.
La microprogrammazione e' la soluzione tipica di architetture CISC per implementare l'unita' di controllo. Questo perche' l'adozione di tale tecnica permette una maggiore flessibilita' nella progettazione, cioe' rende facile modificare le sequenze di micro-operazioni che implementano le istruzioni eseguite dalla CPU, e premette la realizzazione di un vasto numero di istruzioni da parte della CPU.
**IMPLEMENTAZIONE ISTRUZIONI**
Ad ogni codice operativo si fa corrispondere indirizzo di inizio di un µprogramma. Nell'implementazione µprogrammata fase di fetch e fase di execute possono essere descritte in un linguaggio di µprogrammazione. Ad ogni istruzione macchina viene associato un µprogramma che e' formato da una sequenza di µistruzioni. Esse sono formate da micro-ordini (ognuno corrispondente ad un segnale di controllo) registrati nella memoria ROM detta anche memoria di controllo ed organizzati in una word chiamata word di controllo. Nella word di controllo ogni bit corrisponde ad un micro-ordine, ovvero rappresenta una linea di controllo. Le unita' di controllo µprogrammate si possono suddividere in due categorie:
- µprogrammazione orizzontale quando le microistruzioni sono composte da un numero elevato di bit e quindi possono essere svolti molti compiti in parallelo, generando svariati segnali di controllo contemporaneamente;
- µprogrammazione verticale quando le microistruzioni presentano un numero limitato di bit. La microprogrammazione verticale conduce ad una minore velocita' di funzionamento, in quanto, diminuendo il numero di bit, si ha bisogno di piu' microistruzioni per specificare tutte le operazioni svolte con una sola microistruzione orizzontale.
Nell'unita' di controllo µprogrammata il codice operativo dell'istruzione in linguaggio macchina indirizza una locazione della memoria di mapping nella quale e' registrato l'indirizzo di partenza del corrispondente µprogramma. Il µPC contiene l'indirizzo della memoria di controllo e il µIR contiene la microistruzione.

**MULTICORE**

MOTIVAZIONI DI BASE

I microprocessori hanno visto una crescita esponenziale delle prestazioni grazie al miglioramento dell'organizzazione ed all'incremento delle frequenza di clock. Il miglioramento dell'organizzazione del cip e' stato fortemente focalizzato sull'incremento delle parallelismo tra istruzioni. Si e' passati infatti dall'introduzione della pipeline, a CPU superscalari, in cui vi sono pipeline parallele, sino a CPU con multithreading simultaneo (SMT), in cui alle pipeline parallele sono associati banchi di registri replicati. Tali miglioramenti hanno pero' richiesto un aumento di complessita', dalla quale segue quindi una logica piu' complessa quindi difficile da realizzare, progettare e verificare, ed un aumento dell'area del chip per permettere di supportare il parallelismo. Inoltre all'aumentare della densita' del cip e' seguito un aumento esponenziale della potenza richiesta, quindi maggiore energia consumata e maggior calore prodotto. Con le CPU SMT si era giunti al limite della potenza erogabile ed ai limiti del parallelismo a livello di istruzioni, quindi per permettere un aumento delle capacità delle CPU si e' scelto di passare ad architetture multicore. Si ipotizza infatti che con le architetture multicore sia possibile un incremento prestazionale quasi lineare, anche se i vantaggi prestazionali dipendono dallo sfruttamento efficace delle risorse parallele da parte dei programmi (piccole quantita' di codice seriale ha un impatto significativo sulle prestazioni).

POSSIBILI ALTERNATIVE

Organizzazione processore multicore dipende da: numero di core per cip, tipologia di core (se i singoli core sono superscalari o SMT)(core più vecchi avevano organizzazione superscalare, mentre le CPU multicore piu' recenti hanno organizzazione SMT), numero di livelli di cache per chip (che possono essere L1,L2,L3), quantita' di cache condivisa divide i multicore in 4 macrogruppi:

- cache L1 dedicata: ogni core ha la propria cache L1 dedicata, la quale e' suddivisa tra cache dati e cache istruzioni
- cache L2 dedicata: ogni core ha cache L1 ed L2 dedicate
- cache L2 condivisa: ogni core ha cache L1 dedicata, ma vi e' una cache L2 condivisa tra tutti i core
- cache L3 condivisa: ogni core ha cache L1 ed L2 dedicate, ma vi e' una cache L3 condivisa tra tutti i core

L'utilizzo di cache L2 condivisa ha i seguenti vantaggi:

1- interferenza costruttiva: un processo accede alla memoria e carica dei dati. Tali dati servono ad un altro processo su un altro core, il quale trovera' i dati gia' caricati sulla cache condivisa. Vi e' quindi una riduzione accidentale del numero di miss
2- dati condivisi tra piu' core non sono replicati a livello di cache condivisa, ma potrebbero esserlo nella cache non condivisa
3- grazie ad opportuni algoritmi di sostituzione dei blocchi, la cache puo' essere dedicata dinamicamente ad ogni core (quindi processi con minore localita' possono utilizzare più cache)
4- le comunicazione dentro al processore sono piu' facili da realizzare
5- il problema della coerenza dei dati viene confinata nella cache L1

*--manca parte rete sequenziali e reti combinatorie--*