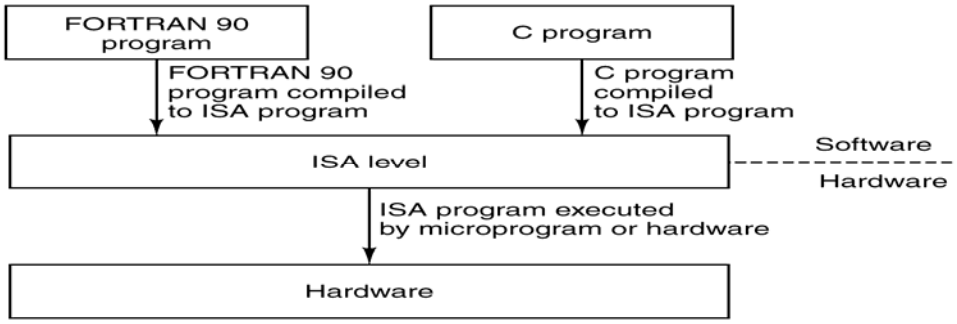
Parte VI

Istruzioni ed indirizzamento

Instruction Set Architecture

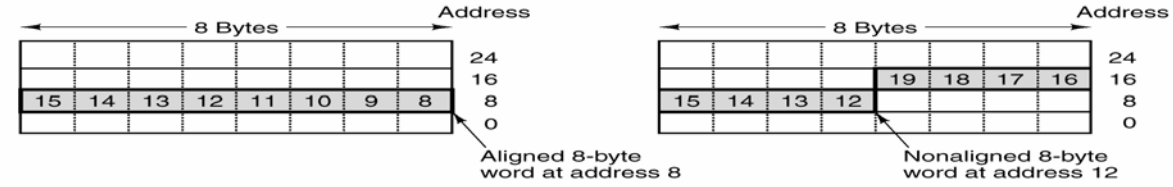


- Il livello ISA è l'interfaccia tra HW e SW
- È il livello più basso a cui il processore è programmabile
- Criteri di scelta:
  - Semplicità di implementazione
  - Efficienza della microarchitettura
  - Semplicità di generazione del codice
  - Compatibilità con il passato

Il livello ISA

- Costituisce il riferimento per coloro che scrivono i compilatori (oppure che programmano in assembler)
- Può essere definita in documenti formali: **Es.** SPARC e JVM
- Può non esistere una definizione formale: **Es**. IA-32 di Intel
- Caratteristiche fondamentali:
  - **Memoria**: organizzazione e modalità di indirizzamento
  - **Registri**: quali registri sono visibili al livello ISA e quali funzioni hanno
  - **Indirizzamento**: modalità con cui le istruzioni fanno riferimento ai propri operandi
  - **Istruzioni**: repertorio delle istruzioni macchina
  - **Modi di funzionamento**: *user mode* e *kernel mode*

Modello della memoria



- *Celle elementari*: **byte** di 8 bit (16 con il codice UNICODE ?)
- *Word*: di 4 o 8 byte
- *Allineamento delle word*: non possono cominciare in indirizzi qualsiasi
- *Ordinamento dei caratteri*: finale grande o finale piccolo
- *Spazio degli indirizzi*:
  - Tipicamente 2^{32} 0 2^{64}
  - A volte separato istruzioni e dati
- *Semantica della memoria*: ordinamento degli accessi (non sempre strettamente seriale)

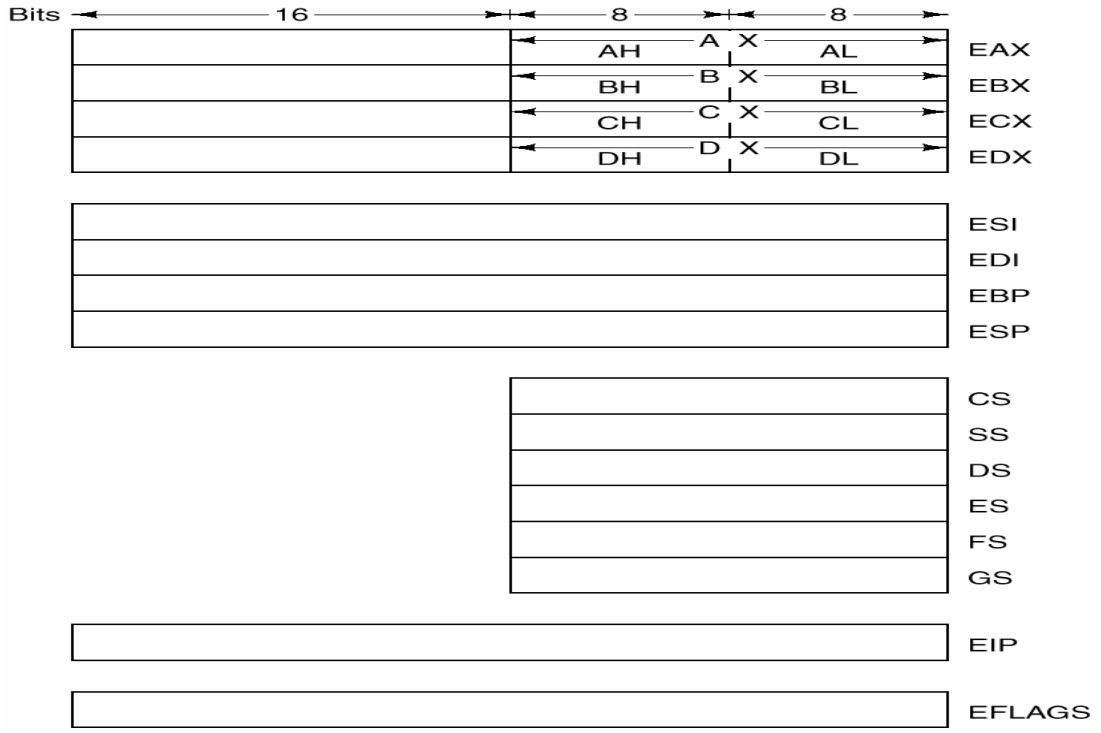
Registri

- Tutti i registri visibili al livello ISA sono visibili anche al livello della μ-architettura, ma non è vero il viceversa
- ***Registri general-purpose***: servono per risultati intermedi e per dati di uso molto frequente
- ***Registri special-purpose***: hanno funzioni specifiche
- ***Registri visibili solo in kernel mode***
- ***PSW (Program Status Word)***: registro che contiene una serie di flag relativi al risultato della ALU (ed altro):
  - **N**: risultato Negativo
  - **Z**: risultato Zero
  - **V**: oVerflow
  - **C**: Carry
  - **A**: Auxiliary carry
  - **P**: Parità del risultato

Livello ISA del Pentium II-IV

- Fortemente influenzato dal vincolo di compatibilità all'indietro
- L'architettura a 32 bit è sostanzialmente la **IA-32** introdotta col 80386
- *Modi di funzionamento*:
  - **Real mode**: si comporta come l'8088
  - **Virtual 8086 mode**: come l'8086 ma *intercetta* tutte le operazioni delicate (es. finestra DOS sotto Windows)
  - **Protected mode**: si comporta come un Pentium II
- *4 livelli di privilegio*: 0=kernel, 3=user
- Spazio di memoria: 16k segmenti di 4GB
- Molti sistemi operativi usano un solo segmento
- È possibile indirizzare il singolo byte
- Word di 4 byte a *finale piccolo*

Registri del Pentium II-IV



Registri del Pentium II-IV (2)

- **EAX**, **EBX**, **ECX**, **EDX**: registri (quasi) general-purpose a 32,16,8 bit:
  - **EAX**: accumulatore
  - **EBX**: puntatori a memoria
  - **ECX**: controllo cicli
  - **EDX**: estende **EAX** a 64 bit nelle divisioni e moltiplicazioni
- **ESI**, **EDI**: puntatori (a stringhe)
- **EBP**: *Base Pointer*, punta alla base dello *stack-frame*
- **ESP**: *Stack Pointer,* punta alla cima dello stack
- **EIP**: *Instruction Pointer*
- **CS**..**GS**: *Registri di Segmento*, puntano ai 6 segmenti (tra i 16k) che sono in uso
- **EFLAGS**: *Program Status Word*

Tipi di dati

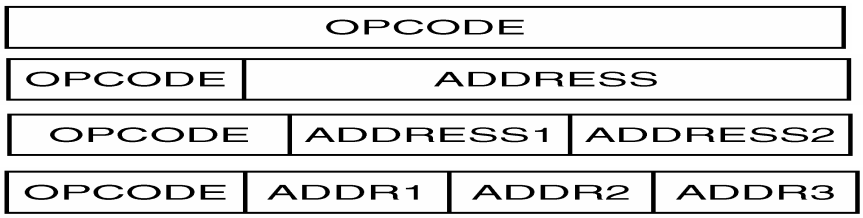
Pentium II-IV

Type	8 Bits	16 Bits	32 Bits	64 Bits	128 Bits
Signed integer	×	×	×		
Unsigned integer	×	×	×		
Binary coded decimal integer	×				
Floating point			×	×	

UltraSPARC II

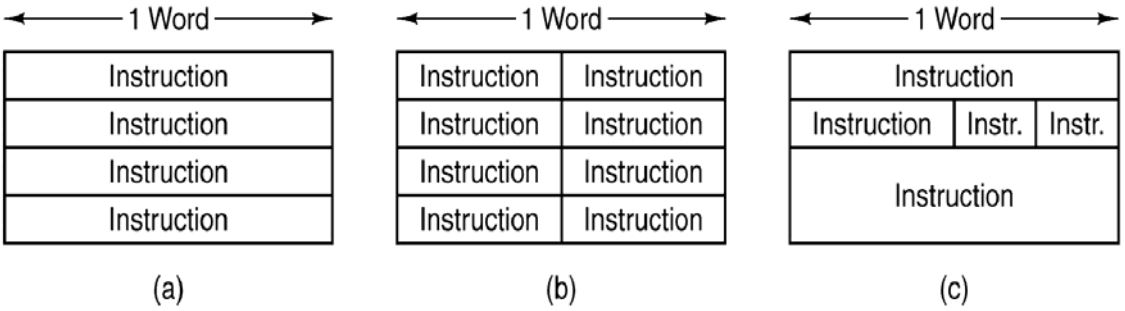
Type	8 Bits	16 Bits	32 Bits	64 Bits	128 Bits
Signed integer	×	×	×	×	
Unsigned integer	×	×	×	×	
Binary coded decimal integer					
Floating point			×	×	×

Formato delle istruzioni



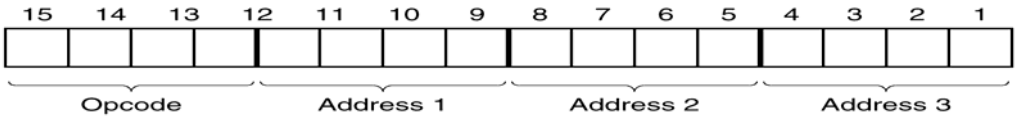
- Lunghezza *fissa* o *variabile*
- Lunghezza fissa semplifica la decodifica
- Istruzioni corte sono preferibili perché riducono la banda di memoria necessaria al fetch delle istruzioni
- Se l'opcode è di k bit si hanno al massimo 2^k istruzioni diverse
- I campi indirizzi possono fare riferimento sia alla memoria che ai registri
- Complesse *modalità di indirizzamento* permettono di indirizzare gli operandi anche con pochi bit

Lunghezza delle istruzioni



- Le istruzioni possono avere lunghezza fissa o variabile
- La lunghezza fissa semplifica il fetch delle istruzioni
- Una word può contenere una o più istruzioni
- Se le istruzioni hanno formato variabile non c'è relazione tra word e istruzioni

Espansione dei codici operativi

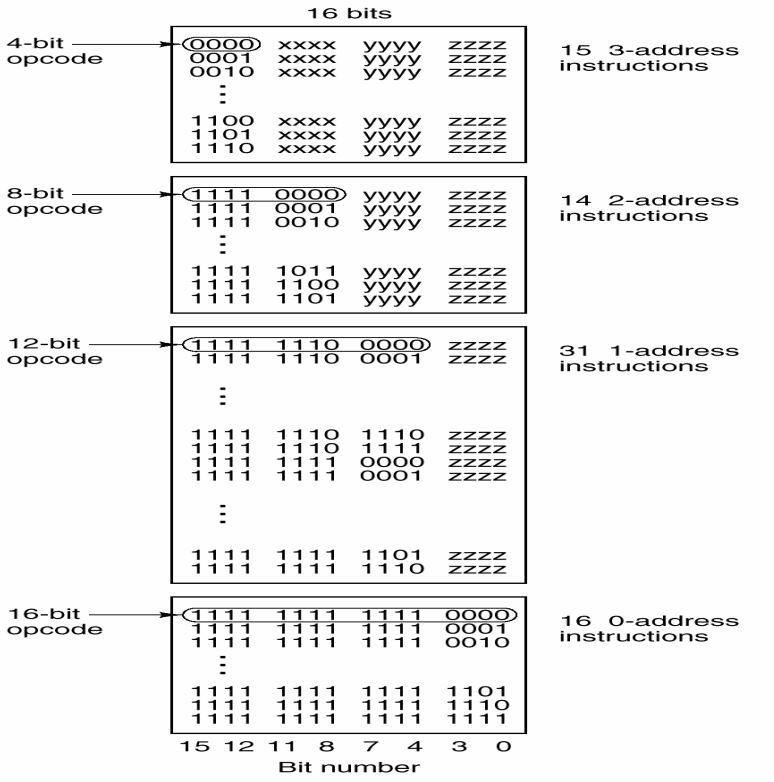


- Il numero di bit dedicati all'opcode non è costante
- Un primo tipo di istruzioni ha un codice operativo corto
- Alcuni valori del codice segnalano che anche i bit successivi ne fanno parte
- Alcuni dei campi indirizzo vengono dedicati all'espansione del codice:

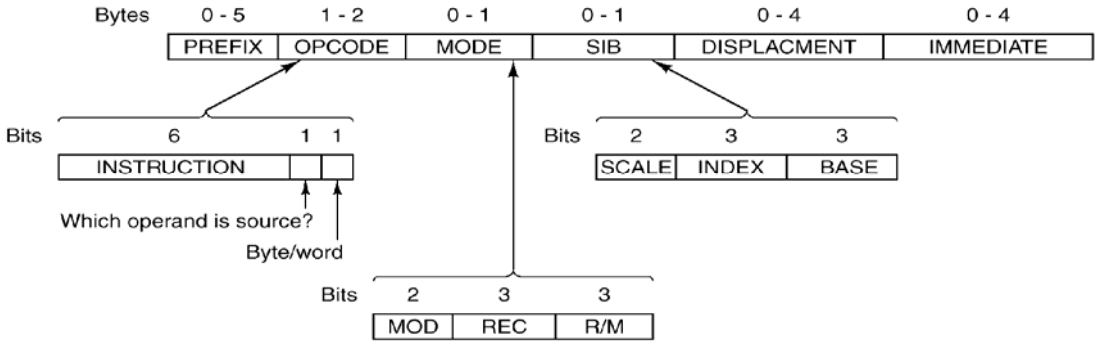
Esempio

- Istruzioni a 3 indirizzi: opcode 4 bit
- Istruzioni a 2 indirizzi: opcode 8 bit
- Istruzioni a 1 indirizzo: opcode 12 bit
- Istruzioni a 0 indirizzi: opcode 16 bit

Espansione dei codici operativi (2)



Formato delle Istruzioni del Pentium II-IV



- Lunghezza delle istruzioni *molto variabile*
- Uno dei due operandi è *sempre un registro*, l'altro può essere sia un registro che in memoria
- MODE stabilisce la modalità di indirizzamento
- L'indirizzo di memoria è l'offset di un segmento, dipende dall'opcode, dai registri usati e dal *prefisso*
- Possibile l'indirizzamento *immediato*: operando nell'istruzione

Modalità di indirizzamento

- **Immediato**: il valore dell'operando è nell'istruzione
- **Diretto**: l'istruzione contiene l'indirizzo di memoria completo dell'operando
- **Indiretto**: l'indirizzo di memoria fornito contiene *l'indirizzo dell'operando*
- **A registro**: si specifica un registro che contiene l'operando (o che lo riceverà)
- **Indiretto a registro**: il registro specificato contiene *l'indirizzo dell'operando*
- **A registro indice**: l'indirizzo è dato da una costante più il contenuto di un registro
- **A registro base**: viene sommato a tutti gli indirizzi il contenuto di un registro
- **A stack**: l'operando è sulla cima dello stack (o ci deve andare)

Indirizzamento indiretto a registro

```
        MOV R1,#0          ; accumulate the sum in R1, initially 0
        MOV R2,#A          ; R2 = address of the array A
        MOV R3,#A+1024 ; R3 = address if the first word beyond A
LOOP: ADD R1,(R2)          ; register indirect through R2 to get operand
        ADD R2,#4          ; increment R2 by one word (4 bytes)
        CMP R2,R3          ; are we done yet?
        BLT LOOP           ; if R2 < R3, we are not done, so continue
```

- Calcola la somma degli elementi di un array di 256 interi che inizia all'indirizzo A
- Ciascun intero occupa 4 byte
- La somma viene accumulata in R1
- R2 punta all'elemento corrente

Indirizzamento a registro indice

```
        MOV R1,#0       ; accumulate the OR in R1, initially 0
        MOV R2,#0       ; R2 = index, i, of current product: A[i] AND B[i]
        MOV R3,#4096 ; R3 = first index value not to use
LOOP: MOV R4,A(R2) ; R4 = A[i]
        AND R4,B(R2)  ; R4 = A[i] AND B[i]
        OR R1,R4        ; OR all the Boolean products into R1
        ADD R2,#4       ; i = i + 4 (step in units of 1 word = 4 bytes)
        CMP R2,R3       ; are we done yet?
        BLT LOOP        ; if R2 < R3, we are not done, so continue
```

- Calcola l'OR di A[i] AND B[i] dove A e B sono due array
- R1 accumula l'OR degli AND
- R2 contiene l'indice corrente sugli array
- R3 contiene la costante 4096, per controllare la fine del loop
- R4 è utilizzato per appoggiare i singoli AND

Indirizzamento a stack

- La stack è utilizzato per:
  - *Appoggiare* risultati intermedi
  - Gestire le chiamate di procedura
  - Calcolare espressioni aritmetiche
- Lo *stack pointer* SP punta all'elemento affiorante dello stack
- Operazioni fondamentali:
  - PUSH: aggiunge un elemento alla cima dello stack
  - POP: preleva un elemento dalla cima dello stack
  - Operazioni aritmetiche sui due elementi affioranti che mettono al loro posto il risultato

Gestione dell'I/O

- Un'operazione di I/O consiste nel trasferimento di dati tra un device di I/O e la memoria
- Tre modi fondamentali di gestire l'I/O

A) *I/O programmato con busy waiting*

La CPU interroga periodicamente i dispositivi (*polling*) e cicla a vuoto durante le attese (*busy waiting*)

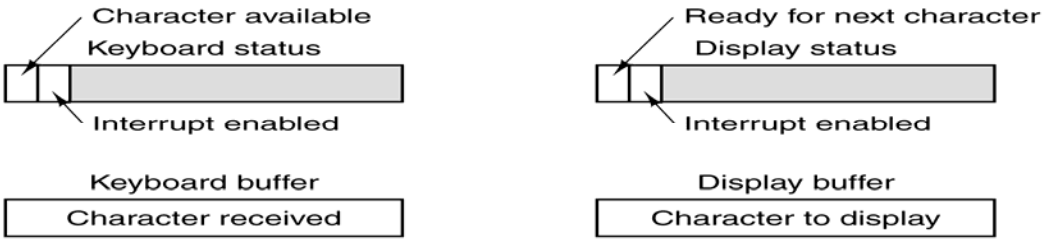
B) *I/O gestito con interruzioni*

La CPU avvia l'operazione di I/O e poi si dedica ad altro fino a che il device non manda una interruzione

C) *DMA (Direct Memory Access)*

La CPU avvia l'operazione poi gestita interamente dal controllore DMA

I/O programmato

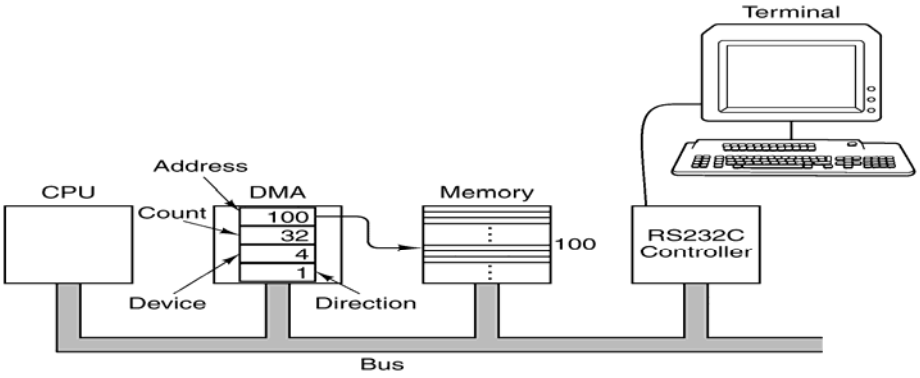


- I *controller* hanno diverse porte che possono essere lette e scritte dalla CPU
- Nei buffer vengono letti o scritti i caratteri scambiati con il controller
- I registri di stato contengono bit che la CPU controlla per sapere se i dati sono disponibili, o possono essere scritti
- Dati i tempi dei dispositivi di I/O il busy waiting comporta un notevole spreco della risorsa CPU
- Usato solo in sistemi molto semplici (o d'antiquariato)

I/O con interrupt

- Esempio: lettura da disco
- La CPU avvia l'operazione di I/O scrivendo le informazioni opportune nelle porte del controller
- La CPU passa all'elaborazione di un altro task
- Il controller avvia e sovraintende allo svolgimento dell'operazione di I/O (posizionamento delle testine ecc.)
- Solo quando i dati sono disponibili il controller interrompe la CPU
- La CPU è direttamente coinvolta nel trasferimento dei dati tra controller, essa legge i dati e li copia in memoria

DMA (Direct Memory Access)



- La CPU programma il controller DMA specificando:
  - Quanti byte trasferire
  - Da quale device
  - A che indirizzi
- Il controller gestisce l'intera operazione
- Il controller DMA può gestire più operazioni contemporaneamente

Istruzioni del Pentium II-IV (1)

Moves

MOV DST,SRC	Move SRC to DST
PUSH SRC	Push SRC onto the stack
POP DST	Pop a word from the stack to DST
XCHG DS1,DS2	Exchange DS1 and DS2
LEA DST,SRC	Load effective addr of SRC into DST
CMOV DST,SRC	Conditional move

Arithmetic

ADD DST,SRC	Add SRC to DST
SUB DST,SRC	Subtract DST from SRC
MUL SRC	Multiply EAX by SRC (unsigned)
IMUL SRC	Multiply EAX by SRC (signed)
DIV SRC	Divide EDX:EAX by SRC (unsigned)
IDIV SRC	Divide EDX:EAX by SRC (signed)
ADC DST,SRC	Add SRC to DST, then add carry bit
SBB DST,SRC	Subtract DST & carry from SRC
INC DST	Add 1 to DST
DEC DST	Subtract 1 from DST
NEG DST	Negate DST (subtract it from 0)

Binary coded decimal

DAA	Decimal adjust
DAS	Decimal adjust for subtraction
AAA	ASCII adjust for addition
AAS	ASCII adjust for subtraction
AAM	ASCII adjust for multiplication
AAD	ASCII adjust for division

Istruzioni del Pentium II-IV (2)

Boolean

AND DST,SRC	Boolean AND SRC into DST
OR DST,SRC	Boolean OR SRC into DST
XOR DST,SRC	Boolean Exclusive OR SRC to DST
NOT DST	Replace DST with 1's complement

Shift/rotate

SAL/SAR DST,#	Shift DST left/right # bits
SHL/SHR DST,#	Logical shift DST left/right # bits
ROL/ROR DST,#	Rotate DST left/right # bits
RCL/RCR DST,#	Rotate DST through carry # bits

Test/compare

TST SRC1,SRC2	Boolean AND operands, set flags
CMP SRC1,SRC2	Set flags based on SRC1 - SRC2

Transfer of control

JMP ADDR	Jump to ADDR
Jxx ADDR	Conditional jumps based on flags
CALL ADDR	Call procedure at ADDR
RET	Return from procedure
IRET	Return from interrupt
LOOPxx	Loop until condition met
INT ADDR	Initiate a software interrupt
INTO	Interrupt if overflow bit is set

Istruzioni del Pentium II-IV (3)

Strings

LODS	Load string
STOS	Store string
MOVS	Move string
CMPS	Compare two strings
SCAS	Scan Strings

Condition codes

STC	Set carry bit in EFLAGS register
CLC	Clear carry bit in EFLAGS register
CMC	Complement carry bit in EFLAGS
STD	Set direction bit in EFLAGS register
CLD	Clear direction bit in EFLAGS reg
STI	Set interrupt bit in EFLAGS register
CLI	Clear interrupt bit in EFLAGS reg
PUSHFD	Push EFLAGS register onto stack
POPFD	Pop EFLAGS register from stack
LAHF	Load AH from EFLAGS register
SAHF	Store AH in EFLAGS register

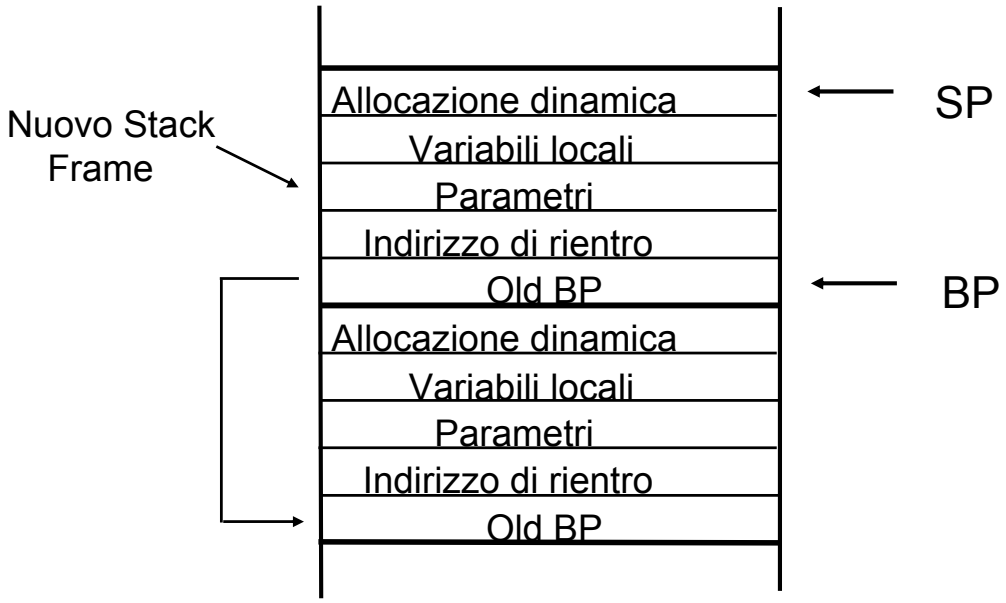
Miscellaneous

SWAP DST	Change endianness of DST
CWQ	Extend EAX to EDX:EAX for division
CWDE	Extend 16-bit number in AX to EAX
ENTER SIZE,LV	Create stack frame with SIZE bytes
LEAVE	Undo stack frame built by ENTER
NOP	No operation
HLT	Halt
IN AL,PORT	Input a byte from PORT to AL
OUT PORT,AL	Output a byte from AL to PORT
WAIT	Wait for an interrupt

Chiamata di procedura

- Per ciascuna chiamata viene allocato sullo stack un nuovo *stack frame*
- Lo stack frame contiene:
  - I parametri in entrata e in uscita
  - Le variabili locali
  - L'indirizzo di rientro
  - Un puntatore allo stack frame del chiamante
- Lo *stack pointer* SP punta alla cima dello stack
- Il *base pointer* BP punta *alla base del frame*
- L'accesso ai parametri e alle variabili locali e avviene tramite offset da BP
- La posizione rispetto a BP è nota a tempo di compilazione e costante
- La posizione rispetto a SP non è costante (possono essere fatte PUSH e POP durante l'esecuzione)
- All'atto del rientro lo stack frame viene deallocato

Struttura dello stack frame



Trap e interrupt

- La *trap* è una procedura automatica che viene iniziata da una condizione eccezionale che si verifica *durante l'esecuzione di un programma*
- Le trap sono *sincrone e dipendenti da quello che succede sulla CPU*, mentre le interruzioni sono *asincrone e nascono all'esterno della CPU*
- Le trap si originano da test fatti a livello del microprogramma
- La gestione delle trap è affidata al trap handler ed è in tutto simile a quella delle interrupt
- Esempi di trap:
  - overflow e underflow
  - violazione di protezione
  - divisione per zero
  - chiamata di sistema

Interruzioni: azioni HW

Quando l'interruzione si origina e viene servita queste azioni preliminari *vengono svolte a livello hardware*:

1) Il controller genera l'interruzione

2) La CPU, quando è pronta a servirla, alza il segnale di *aknowledge*

3) Quando il controller vede l'aknowledge risponde mettendo sul bus il *vettore di interruzione*

4) La CPU legge e salva il vettore di interrupt

5) La CPU salva il PC (Program Counter) e la PSW (Program Status word) sullo stack

6) La CPU individua, per il tramite del vettore di interruzione, l'indirizzo iniziale della routine che serve l'interruzione e lo carica nel PC

Interruzioni: azioni SW

Inizia ora l'esecuzione della routine di servizio che svolge le seguenti azioni:

7) Salva sullo stack i registri della CPU

8) Individua il numero esatto del device

9) Legge tutti i codici di stato ecc.

10) Gestisce eventuali errori di I/O

11) Legge (o scrive) i dati e incrementa i conteggi

12) Se necessario informa il device che il servizio dell'interruzione è concluso

13) Ricarica tutti i registri salvati sullo stack

14) Esegue un'istruzione di RETURN FROM INTERRUPT ripristinando lo stato della CPU precedente l'interruzione

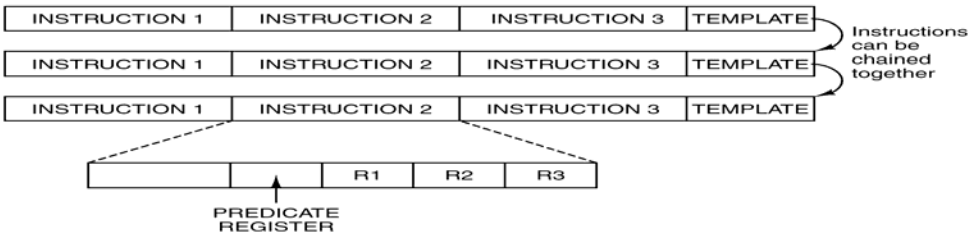
I problemi del Pentium II

- *IA-32 è irrimediabilmente CISC*: le sue istruzioni possono essere spezzate in istruzioni RISC ma questo richiede tempo e spazio su chip
- *Indirizzamento orientato a memoria*
- *Pochi registri e asimmetrici*: molti risultati intermedi devono essere appoggiati in memoria
- *Pochi registri=molte dipendenze*: rende difficile l'esecuzione parallela di più istruzioni
- *Necessita di una pipeline lunga*: rende difficile la predizione dei salti
- *Rimedia con l'esecuzione speculativa* ma di fatto crea altri problemi
- *4 GB di spazio di indirizzamento*: ormai poco per un grosso server

L'architettura IA-64

- Dopo aver spremuto fino in fondo la IA-32, Intel sta pensando seriamente di rompere con il passato e di proporre una nuova ISA: la IA-64
- La IA-64 è una architettura a 64 bit sviluppata in collaborazione con HP
- Disegno basato in parte sull'architettura PA-RISC di HP
- La prima implementazione è una CPU di fascia alta denominata in codice *Merced*
- Tanto per non parlare di compatibilità all'indietro Merced è un processore dual-mode e può eseguire anche il vecchio codice IA-32

Il modello IA-64



- *EPIC (Explicitly Parallel Instruction Computing)*
- Si cerca di evidenziare le possibilità di esecuzione parallela delle istruzioni
- Le istruzioni vengono in *bundle* cioè in gruppi di tre, che possono essere a loro volta *incatenati*
- La parte *template* fornisce informazioni sulle possibilità di esecuzione parallela
- Molto del lavoro è spostato al tempo di compilazione ed ottimizzazione

Esecuzione condizionale

```
if (R1 == 0)          CMP R1,0        CMOVZ R2,R3,R1
    R2 = R3;          BNE L1
                      MOV R2,R3
                L1:
```

(a) (b) (c)

- Si cerca di diminuire i salti condizionati con la tecnica della *predication*
- La predication è un'estensione del concetto di *esecuzione condizionale*
- Dato il codice sorgente (a) la traduzione classica tramite salto condizionato è data in (b)
- La traduzione in (c) sfrutta un'istruzione ad *esecuzione condizionale* e non contiene salti condizionati

Esecuzione condizionale (2)

```
if (R1 == 0) {          CMP R1,0        CMOVZ R2,R3,R1
    R2 = R3;            BNE L1          CMOVZ R4,R5,R1
    R4 = R5;            MOV R2,R3       CMOVN R6,R7,R1
} else {                MOV R4.R5       CMOVN R8,R9,R1
    R6 = R7;            BR L2
    R8 = R9;        L1: MOV R6,R7
}                       MOV R8,R9
                    L2:
     (a)                    (b)                (c)
```

- Nella versione condizionale (c) il codice è costituito da un unico blocco basico senza salti
- L'unico vincolo per l'esecuzione di ciascuna istruzione è la conoscenza della condizione

Esecuzione predicativa

```
if (R1 == R2)           CMP R1,R2       CMPEQ R1,R2,P4
    R3 = R4 + R5;       BNE L1          <P4> ADD R3,R4,R5
else                    MOV R3,R4       <P5> SUB R6,R4,R5
    R6 = R4 – R5        ADD R3,R5
                        BR L2
                    L1: MOV R6,R4
                        SUB R6,R5
                    L2:
     (a)                    (b)                (c)
```

- La prima istruzione stabilisce il valore del registro predicativo P4, e mette P5 al valore negato
- L'esecuzione delle istruzioni successive dipende dai valori di P4 e P5
- Istruzioni predicative possono andare in pipeline senza problemi di stallo

Intel Itanium 2

- Tre livelli di cache (sul chip)
  - L1: 16+16 KB
  - L2: 256 KB
  - L3: fino a 6 MB
- Spazio di indirizzamento
  - Indirizzi virtuali a 64 bit
  - Indirizzi fisici a 50 bit
  - Pagine fino a 4 GB
- Esecuzione di due bundle (6 istruzioni) per ciclo di clock
- 328 registri sul chip
- Banda verso il system bus &.4 GB/s

Architettura AMD a 64 bit (x86-64)

- AMD ha finora prodotto processori a 32 bit completamente compatibili con la piattaforma IA-32
- Essi possono eseguire lo stesso set di istruzioni, e quindi lo stesso codice, anche se hanno soluzioni architetturali diverse
- Con il passaggio ai 64 bit *le strade divergono*
- AMD propone la tecnologia x86-64, a 64 bit ma basata su IA-32, con i processori *Opteron* e *Athlon-64*
- X86-64 è compatibile con il vecchio codice, ma incompatibile con la piattaforma IA-64 di Intel
- AMD si offre come soluzione più vantaggiosa nella fase di transizione
- Sia Microsoft che Red Hat hanno assicurato la realizzazione di versioni dei loro sistemi operativi per la piattaforma x86-64

Modi di funzionamento di Opteron

- *Legacy mode*
  - Esegue direttamente i sistemi operativi e le applicazioni a 32 bit, senza necessità di ricompilarle
  - Può rimpiazzare direttamente un processore IA-32
- *Compatibility mode*
  - Permette di eseguire le applicazioni a 32 bit sotto un sistema operativo a 64 bit
  - Possibile l'upgrade del SO senza toccare le applicazioni
- *New-generation mode*
  - Permette di eseguire le applicazioni a 64 bit sotto un sistema operativo a 64 bit
  - Occorre evidentemente ricompilare le applicazioni