Assembly

Programmazione in linguaggio macchina (o meglio in assembly):
programmare utilizzando istruzioni direttamente eseguibili dal processore.

Questa parte del corsi si accompagna a lezioni in laboratorio: programmazione in assembly con uso di un simulatore.

Motivazioni

- Programmare in assembly aiuta a capire funzionamento e meccanismi base di un calcolatore.
- Fa meglio comprendere cosa accade durante l'esecuzione di un programma scritto ad alto livello.
- Insegna una metodologia, si acquisiscono delle abilità.
- In alcuni casi è utile programmare in assembly.

Programmazione Assembly

Vantaggi

controllo dell'hardware: si definiscono esattamente le istruzioni da eseguire e le locazioni di memoria da modificare.

Utile per:

- gestione hardware, risorse del computer (kernel, driver)
- ottimizzare il codice: programmi più efficienti.

Programmazione Assembly

Svantaggi:

- scarsa portabilità: i programmi ad hoc per una famiglia di processori,
- scomodi: istruzioni poco potenti, programmi lunghi,
- facile cadere in errore: programmi poco strutturati e leggibili.
- compilatori sempre più sofisticati producono codice efficiente (soprattutto per le architetture parallele), è difficile fare meglio programmando in assembly.

Processore scelto: MIPS

Tanti linguaggi macchina quante le famiglie di processori: IA-32 Intel Core, PowerPC (IBM, Motorola), Sparc, Arm, ...

Linguaggi simili a grandi linee, ma nel dettaglio con moltissime differenze.

In questo corso: processore MIPS, uno dei primi processori RISC (John L. Hennessy, Stanford University, 1984): poche istruzioni semplici e lineari.

Processore MIPS

Una famiglia di architetture (Instruction set architecture) (e processori): dopo il primo processore a 32-bit, altri a 64-bit, MIPS-3D (con istruzioni vettoriali).

Usato in:

- workstation: per alcuni anni da Silicon Graphics poi sostituito da IA-32,
- video-giochi: PS2, PSP, Nintendo 64
- sistemi embedded: stampanti, DVD, router, ADSL modem, TiVo, WindowsCE ....

Possibili alternative

- IA-32 del Core (Pentium). Linguaggio difficile da imparare e da usare. Un aggregato di istruzioni costruito nell'arco degli anni. Problemi di legacy (compatibilità con linguaggi precedenti).
- Assembly 8088. Descritto nel libro di testo. Piuttosto vecchio, progenitore del IA-32, diverso dai linguaggi macchina attuali.
- ARM Più complesso. Manca un semplice simulatore multipiattaforma.

MIPS

Vantaggi:

- estremamente semplice e elegante, facile da imparare,
- molto usato in didattica,
- simile agli altri RISC.

Svantaggi:

- programmi lunghi: istruzioni semplici fanno cose semplici.
  alcuni assembly (Motorola 6800) che permettono una scrittura più sintetica dei programmi.

Riferimenti

In inglese e reperibili nella pagina web del corso.

- *Sparse Notes on an MIPS Processor Architecture and its Assembly Language*
  Appunti di Guido Sciavicco, presentazione sintetica del linguaggio.
- *A*ssembler, Linkers, and SPIM Simulator
  Appendice al testo: Patterson and Hennessy, *Computer Organization and Design*
  Manuale SPIM, presentazione del linguaggio MIPS.

Simulatore

Necessario per eseguire i programmi, controllarne il comportamento, correggerli.

- SPIM (XSPIM) : simulatore storico, ma non pù supportato.
- MARS: sviluppato dalla Missouri University, scritto in Java (codice universale se abbinato ad una JPSE)
  parole chiave: ‘mars mips simulator

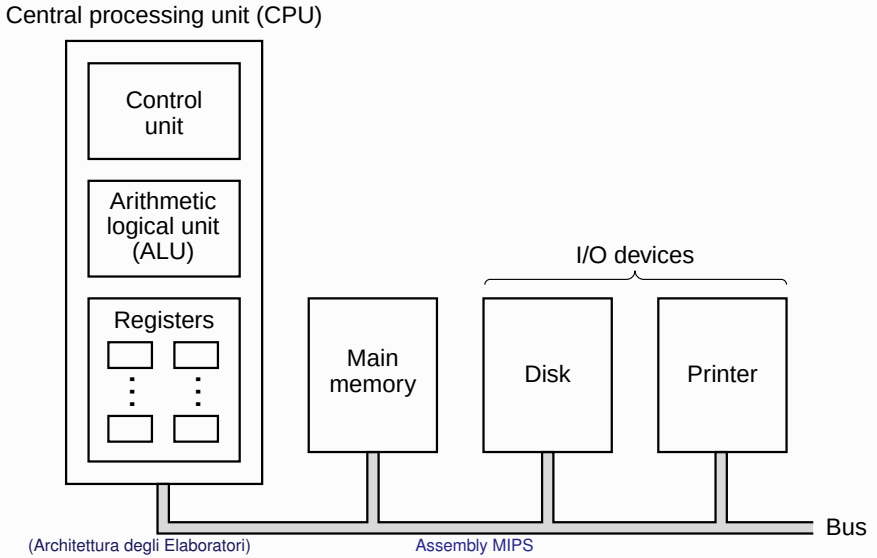
Struttura schematica di un calcolatore

Le istruzioni di un linguaggio macchina agiscono direttamente sul hardware.

Fanno riferimento ad un modello di calcolatore.

Mostriamo a grandi linee questa modello.

Struttura schematica di un calcolatore



Semplificazione rispetto al hardware

Dal punto di vista dell'assembly, il calcolatore si compone di principalmente di:

- Processore, con un certo numero di registri.
- Memoria principale.

Nella realtà, l'hardware è piuttosto complesso, questa complessità non direttamente visibile dall'assembly.

Semplificazione rispetto al hardware

Elementi dell'hardware nascosti all'assembly.

- Pipeline, processori superscalari.
- Processore con registri ombra, più registi di quelli effettivamente visibili al programmatore,
- Memoria cache.
- Memoria principale diversa da quella modellata. Memoria virtuale.
- Periferiche.

Nel corso verranno presentati questi aspetti..

Struttura della memoria

Istruzioni assembly fanno riferimento a:

- una memoria, in cui sono presenti dati e codice da eseguire,
- un insieme di registri, contenuti nel data path, argomento e destinazione delle operazioni aritmetiche logiche

MIPS

- Memoria principale: composta da 2^{32} locazioni, della dimensione di un byte. Ogni locazione individuata da un indirizzo di 32 bit.
- Registri processore: 32 registri *generici* di 4 byte (32 bit), `$0, ...$31`.
  Rappresentazione simbolica alternativa: `$s0 ...$s7, $t0 ...,$t9, $v0, $v1, $a0 ...$a3, $sp, $ra, $zero, ....`
  Le lettere indicano l'uso tipico del registro.
- altri registri *specializzati*.

Operazioni aritmetiche in MIPS

Agiscono sui registri (e non sulla memoria), ogni operazione ha tre argomenti.
Add, subtract, add immediate

- `add $t1, $s1, $s3`: scrive nel registro `$t1` la somma tra (il contenuto di) `$s1` e `$s3`
- `sub $s3, $t0, $s3`: scrive in `$s3` la differenza tra `$t0` e `$s3`
- `addi $t1, $s1, 5`: scrive in `$t1` la somma tra `$s1` e `5`

Operano su numeri interi, in complemento a due.

Formato operazioni

Le istruzioni MIPS hanno 4 possibili formati: modi per specificare argomenti e parametri.

Molte operazioni usano formati con tre argomenti:

`add d, a1, a2,`

- "`d`", registro in cui depositare il risultato,
- "`a1`", registro da cui prelevare il primo argomento dell'operazione
- "`a2`", secondo argomento dell'operazione, può essere un registro, o una costante numerica

Esercizi:

Scrivere pezzi di codice MIPS che inseriscano in `$s1` il valore delle espressioni:

- `$s1 = $s1 + $s2 + $s3`,
- `$s1 = $s1 - $s2 - 3`,
- `$s1 = $s1 + $s2 - 3`,
- `$s1 = 4 × $s2`,
- `$s1 = - $s2`.

Pseudo registro: il registro `$zero` (`$0`), contiene sempre 0.

Assembly e linguaggio macchina

Linguaggio assembly: notazione mnemonica usata dal programmatore per scrivere, analizzare e rappresentare programmi in linguaggio macchina.
Linguaggio macchina: notazione utilizzata dal calcolatore per memorizzare ed eseguire programmi, ogni istruzione è rappresentata mediante una sequenza di bit.

Assembly e linguaggio macchina

La notazione: `sub $s3, $t0, $s3` è assembly: notazione mnemonica di istruzione macchina, sequenza di bit usati nel calcolatore (linguaggio macchina):

`sub`	`$t0`	`$s3`	`$s3`			
0	8	19	19	0	34	.
000000	00100	01011	01011	00000	10010	

Esiste una corrispondenza 1 a 1 tra istruzioni assembly e istruzioni macchina.

In MIPS, ogni istruzione macchina utilizza 32 bit.

Formati del linguaggio macchina

L'istruzione `add rd, rs, rt`, in linguaggio macchina:

`opcode` -	`rs` -	`rt` -	`rd` -	`sh arg` -	`fun code`
6 bit -	5 bit -	5 bit -	5 bit -	5 bit -	6 bit

- opcode: operation code,
- rs - rt : registri argomento,
- rd: registro destinazione,
- shift argument: parte usata dalle istruzione di shift,
- function code: specifica ulteriormente il tipo di operazione.

Secondo formato

L'istruzione `addi rd, rs, imm`, in linguaggio macchina:

`opcode` -	`rs` -	`rd` -	`imm`
6 bit -	5 bit -	5 bit -	16 bit

`imm`: argomento immediato, un numero di 16 bit

Pseudo-istruzioni

Semplificare la programmazione, l'assembly MIPS contiene istruzioni extra rispetto a quelle del linguaggio macchina.

Pseudo-istruzioni vengono tradotte in sequenze di (1-4) istruzioni macchina.

Le pseudo-istruzioni utili e naturali. In programmazione, quasi indistinguibili dalle istruzioni.

Non implementate dal processore, non inserite nel linguaggio macchina, per motivi di efficienza.

Istruzioni trasferimento dati

Solo 32 registri: necessario usare la memoria principale.

- load word - da memoria a registro
  `lw $t0, 4($s1)`
  copia nel registro `$t0` 4 byte a partire dall'indirizzo.
  `4 + $s1`, (base e offset).

base: numero a 16 bit, base $< 2^{16}$
(tutta listruzione memorizzata in 32 bit)
Limite superabile mediane pseudo-istruzioni.

Altre modalità di indirizzamento

mediante pseudo-istruzioni :

- `lw $t0, ($s1)`,
  equivalente a `lw $t0, 0($s1)`
- `lw $t0, 0x1122`,
  equivalente a `lw $t0, 0x1122($zero)`
- `lw $t0, 0x11223344`,
  equivalente a ...

Nota: valori costanti scritti mediante

- la notazione decimale (es. 11224455),
- o la notazione esadecimale (es. 0*x*11224455).

Parole allineate

lw legge 4 byte consecutivi in memoria (per comporre una parole),
legge solo a parole allineate, ossia lindirizzo del primo byte é un multiplo di 4,

MIPS, big-endian: si leggono 4 byte alla volta, bit meno significativi negli indirizzi grandi.

Per inserire dati in memoria:

- store word - da registro a memoria
  `sw $s0, 4($t3)`

Trasferire parti di una parola

Dalla memoria è possibile leggere o scrivere anche:

- un singolo byte, sequenza di 8 bit:
  load byte `lb`
  load byte unsigned `lbu`
  store byte `sb`,
- un mezza parola (half word) sequenza di 16 bit, 2 byte :
  load half word `lh`
  load half word unsigned `lhu`
  store half word `sh`.

Inserire una costante in un registro

- load immediate - una costante in un registro
  `li $t1, 0x11224455`,
  scrive in `$t1` il numero 11224455_{16}.
  Pseudo-istruzione, tradotta nella coppia di istruzioni:
  1. Load upper immediate,
     `lui $t1, 0x1122`, scrive `0x1122` nei 16 bit più significativi di `$t1`, azzera i 16 meno significativi.
  2. + l'istruzione `...addi $t1, $t1, 0x4455`

Trasferimento dati

- move pseudo-istruzione per trasferire dati da registro a registro
  `move $t0, $t1` : copia in `$t0` il contenuto del registro `$t1`

Con quale (singola) istruzione può essere tradotta?
Proporre 4 possibili alternative.

Vettori

Tipi di dati complessi.
Sequenza di valori dello stesso tipo.
Es. Una coordinata cartesiana nello spazio $\langle 0, 3, 2 \rangle$

Supportati dai linguaggi alto livello.
Implementati *a mano*, in assembly.

- allocati in memoria principale, in locazioni consecutive;
- si accede, identifica, il vettore mediante l'indirizzo base.

Vettori

I diversi elementi di uno stesso vettore sono identificati da un indice.

Per operare sull'elemento *i*-esimo del vettore bisogna:

- valutare la sua posizione in memoria:
  = "indirizzo base" + "displacement"
  "displacement" := "indice" $\times$ "dimensione elemento".
- leggerlo o scriverlo con `lw, sw`

Esercizio

Scrivere in `$s1` la somma dei primi tre elementi del vettore (di interi) con indirizzo base `$s0`

```
lw  $s1, ($s0)
lw  $t0, 4($s0)
add $s1, $s1, $t0
lw  $t0, 8($s0)
add $s1, $s1, $t0
```


Esercizio

Scrivere in `$s1` il valore A[`$s2`] + A[`$s2` + 1] + A[`$s2` + 2] del vettore con indirizzo base `$s0`, lindirizzo base del vettore `A` è contenuto in `$s0`.

```
add $t0, $s2, $s2
add $t0, $t0, $t0    # $t0 = 4 × $s2
add $t0, $s0, $t0   # $t0 = indirizzo A[$s2]
lw  $s1, ($t0)
lw  $t1, 4($t0)
add $s1, $s1, $t1
lw  $t1, 8($t0)
add $s1, $s1, $t1
```


Operazioni aritmetiche

multiplication:

`mult $s0, $s1`

Calcola il prodotto tra `$s0` e `$s1`, il risultato, potenzialmente di 64 bit, in due registri, da 32 bit, riservati: `high` e `low`.

Per avere il risultato in registri generici:

move from high:

`mfhi $t0`

move from low:

`mflo $t0`

Più intuitiva e facile da usare la pseudo-istruzione:

`mul $t0, $s0, $s1`

Divisione

division:

`div $s0, $s1`

Mette in `low` il quoziente e in `high` il resto della divisione tra `$s0` e `$s1`.

Pseudo-istruzioni:

```
div $t0, $s0, $s1
rem $t0, $s0, $s1
```

Esercizi:

- scrive le sequenze corrispondenti;
- determinare se `$s0` è pari, un multiplo di 3, risultato in `$s1`.

Istruzioni per controllo di flusso

Programmi sono più complessi di una semplice lista di istruzioni, devono eseguire istruzioni diverse in base ai dati.

Nei linguaggi ad alto livello:

- test (`if-then-else`)
- cicli (`while, repeat`)

In assembly, meccanismi di controllo del flusso elementari e poco strutturati:

- salti condizionati
- salti incondizionati

Salto condizionato

branch equal

`beq $s0, $s1, -17`

se `$s0` = `$s1`, salta a `17` istruzioni più indietro, altrimenti prosegue con l'istruzione successiva.

Lassembly permette di creare un associazione etichetta – indirizzo, istruzione

`beq $s0, $s1, label`

se il test ha successo, salta all'istruzione indicata dall'etichetta `label`,

```
...
label :  add $s0, $s1, $s1
```


Esempi, istruzione di test

`if (i == j) then i = i + 1 else i = j fi`

traduzione: `i, j` $\Rightarrow$ `$s1, $s2`

```
        beq $s1, $s2, then
        move $s1, $s2
        b fine
then:   addi $s1, $s1, 1
fine:
```


Schema generale di traduzione

Una generica istruzione di if-then-else

`if Bool then Com1 else Com2 fi`

viene tradotta in:

```
        branch Bool, then
        Com2
        b fine
then:   Com1
fine:
```

Schema generale

Una generica istruzione di if-then

`if Bool then Com fi`

viene tradotta in:

```
        branch not Bool, fine
        Com
 fine:
```


Esempi, cicli

`while (i != 0) do i = i - 1, j = j + 1 od`

traduzione: `i, j` $\Rightarrow$ `$s1, $s2`

```
 while:  beq $s1, $zero, fine
         addi $s1, -1
         addi $s2, 1
         b while
 fine:
```


Schema traduzione dei cicli

Una generica istruzione di ciclo

`while Bool do Com od`

viene tradotta secondo lo schema:

```
 while:  branch not Bool fine
         Com1
         b while
 fine:
```


Formato macchina istruz. di salto

Indirizzo a cui saltare specificata mediante un numero intero.

`beq, rs, rt, label`

opcode -	rs -	rt -	offset
6 bit -	5 bit -	5 bit -	16 bit

Numeri negativi salti all'indietro.

Salto incondizionato

jump

`j Label`

Salta all'indirizzo (assoluto) indicato da `Label`

jump register

`jr $s0`

Salta all'indirizzo contenuto nel registro `$s0`.

- jump: indirizzo assoluto (salto ad un procedura)
- branch: indirizzo relativo (salto interno ad una procedura)

Altri salti condizionati

branch not equal

`bneq $s0, $s1, label`

branch greater equal (pseudo-istruzione)

`bge $s0, $s1, Label`

branch less than (pseudo-istruzione)

`blt $s0, $s1, Label`

...

implementabili con l'istruzione:

set less than

`slt $s0, $s1, $s2`

assegna `$s0 = 1` se `$s1 < $s2`,
`$s0 = 0` altrimenti.

Altri salti condizionati

Branch on greater than zero

`bgtz $s0, label`

Salta all'etichetta `label` se in registro `$s0` è strettamente maggiore di 0.

Branch on greater equal zero

`bgez $s0, label`

Branch on less than zero

`bltz $s0, label`

Branch on less equal zero

`blez $s0, label`

Esercizi

- Calcolare, e scrivere nel registro `$v0`, il fattoriale del numero contenuto nel registro `$a0`.
- Calcolare, e scrivere nel registro `$v0` la somma degli elementi di un vettore V, di 10 elementi, con indirizzo base il valore di `$a0`,

Branch - Jump

- Branch: indirizzo relativo: 16 bit (+ 2, vengono aggiunti 2 ultimi bit uguali a 0, istruzioni contenute in parole allineate).
  Interno alla stessa procedura, pezzo di codice (semplifica la rilocazione del codice).
- Jump: indirizzo assoluto: 26 bit (+ 2, concatenati ai 4 bit più significativi del PC).
  Salto a procedure esterne.
- Jump register permette di saltare ad indirizzi arbitrari.

Sintassi

Un programma assembly non solo sequenza di istruzioni ma anche:

- etichette `item:` permettono di associare un nomi ad un indirizzo.
- direttive `.text` indicazioni al compilatore (assembler)
  le direttive sono parole precedute da un punto, es. `.data`, `.globl`

Direttive principali

- `.text` quello che segue è il testo del programma.
- `.data` quello che segue sono dati da inserire in memoria
- `.globl` rende l'etichetta che segue visibile agli altri pezzi di codice

Direttive rappresentazione dati

Specificano che tipo di dati inserire in memoria:

- `.word 34, 46, 0xac1`
  ogni numero intero scritto con 4 byte
- `.byte 45, 0x3a`
  ogni numero intero scritto con un byte
- `.ascii "del testo tra parentesi"`
  i codici ascii un dei caratteri della stringa,
- `.asciiz "altro esempio"`
  si aggiunge un byte 0 per marcare fine stringa.

Eventuali etichette usate per dare un nome alla locazione di memoria contenente i dati.

Esempio di programma

```
.data
primes:   .word 2, 3, 5, 7, 11
string :  .asciiz "alcuni numeri primi"

.text
main:     la $t0, primes
          lw $s0, 0 ($t0)
          lw $s1, 4($t0)
          la $t1, string
          lbu $s4, 0 ($t1)
          lbu $s5, 1 ($t1)
```


Specifica di un indirizzo

Nelle istruzioni macchina load `lw`, `lb` e store `sw`, `sb`, l'indirizzo specificato come:
immediato-registro — `lw $a0, 32($s0)`
Esistono pseudo-istruzioni con altre alertmodalità di indirizzamento:

1. registro — `lw $a0, ($s0)`
2. immediato — `lw $a0, 32000`
3. etichetta — `lw $a0, vet1`
4. etichetta + immediato — `lw $a0, vet1+32`
5. etichetta + immediato(registro) — `lw $a0, vet1+32($s0)`

Simulatori: MARS, SPIM

Eseguono i programmi assembly. In generale, permettono di:

- scrivere codice assembly (solo MARS);
- caricare programmi assembly;
- compilare (controllo la correttezza sintattica, generazione codice in linguaggio macchina);
- eseguire il codice, eventualmente passo passo;
- mostrare contenuto di memoria e registri;
- forniscono semplici chiamate al sistema operativo (le stesse per i due simulatori).

Reperibili in rete, parole chiave: assembly, mips, Mars (SPIM)
`http://courses.missouristate.edu/KenVollmar/MARS`

Procedure, funzioni

Meccanismo base della programmazione.

- Struttura il programma in parti (procedure e funzioni).
- Scrittura modulare dei programmi.
- Riutilizzo del codice.
- Necessarie per gestire la complessità dei programmi.

Utilizzate in programmazione assembly.

Supportate dai linguaggi macchina, ma non immediatemente disponibili:
una chiamata di procedura corrisponde ad una sequenze di istruzioni macchina.

Procedure, funzioni

Una chiamate di procedura comporta:

- passare controllo ad altro codice,
- passare i parametri,
- allocare spazio di memoria variabili locali,

L'uscita una procedura comporta:

- recuperare lo spazio in memoria,
- ritornare controllo (e dati) al chiamante.

Implementati con meccanismi elementari nel MIPS.

Mostriamo come le chiamate di procedura vengono tradotte in linguaggio macchina.

Chiamata di procedure

Necessità di ricordare indirizzo di ritorno:

Jump and link

`jal Label`

salta all'etichetta `Label`, salva nel registro `$ra` in return address, (`$31`)
l'indirizzo dell'istruzione successiva (a `jal Label`).

Uscita dalla procedura:

Jump register

`jr $ra`

Altre istruzione di salto con link:

`jalr, bgezal, bltzal`

Esempio: funzione fattoriale

```
.text
main:       ...
            jal fattoriale
            ...
fattoriale: ...
            ...
            jr $ra
fibonacci:  ...
            ...
```


Passaggi dei parametri - risultati

Si usano i registri per il passaggio dei parametri (pochi). Convenzione sull'uso dei registri:

- Registri `$a0, ..., $a3` (`$4, ..., $7`) utilizzati per passare argomenti ad una procedura-funzione.
- Registri `$v0, $v1` (`$2, $3`) utilizzati per restituire, al programma chiamante, i valori risultato di una funzione.

Eventuali parametri extra si passano attraverso la memoria.

Esempio: funzione fattoriale

```
.text
main:       ...
            li $a0, 5
            jal fattoriale
            ...
fattoriale: ...
            jeqz $a0, end
            ...
            mflo $v0
            jr $ra
fibonacci:  ...
            ...
```


Salvataggio dei registri

Programma principale e procedura agiscono sullo stesso insieme di registri (processore unico).
Bisogna evitare interferenze improprie.

```
main:       ...
            add $s0, $s0, 1
            li $a0, 5
            jal fattoriale
            ...
fattoriale: ...
            ...
            move $s0, $a0
            ...
            jr $ra
```


Prima soluzione

Registri non modificabili dalle procedure.
Una procedura, come primo passo, salva in memoria i registri che dovrà modificare. Prima di tornare il controllo al programma principale, ne ripristina il valore.

Esempio prima soluzione

```
main:       ...
            add $s0, $s0, 1
            li $a0, 5
            jal fattoriale
            ...
fattoriale: ...
            sw $s0, ($sp)
            ...
            move $s0, $a0
            ...
            lw $s0, ($sp)
            jr $ra
```

Seconda soluzione

Registri modificabili dalle procedure. Programma principale prima di chiamare una procedura, salva in memoria i registri che deve preservare. Terminata la chiamata li ripristina.

Esempio seconda soluzione

```
main:       ...
            add $s0, $s0, 1
            sw $s0, ($sp)
            li $a0, 5
            jal fattoriale
            lw $s0, ($sp)
            ...
fattoriale: ...
            ...
            move $s0, $a0
            ...
            jr $ra
```


Convenzioni sui registri

In MIPS si usano entrambi gli approcci.

- Registri `$s0, ..., $s7, $ra, $sp` non modificabili dalla procedura chiamata. Se scritti, si deve ripristinarne il valore iniziale.
- Registri `$t0, ..., $t9, $a0, ...$a3, $v0, $v1` modificabili liberamente dalla procedura chiamata.

Convenzione: permette di integrare più facilmente codice macchina scritto da persone diverse.

Allocazione spazio di memoria

Ogni procedura necessita di uno zona di memoria per:

- variabili locali;
- copia dei registri;
- parametri e risultati.

Questa zona di memoria allocata in uno stack.

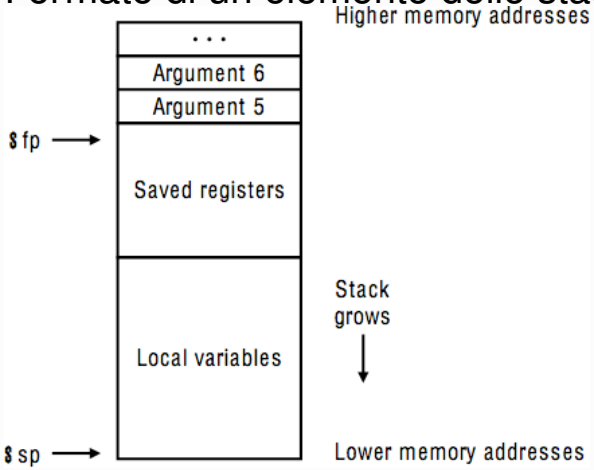
Una pila di zone di memoria consecutive:

- Chiamata di procedura: allocare spazio.
- Uscita dalla procedura: recuperare spazio.

Last In - First Out: la procedura chiamata più recentemente è la prima a terminare.

Stack chiamate di procedura

Formato di un elemento dello stack:



Implementazione dello stack

Risiede in una zona fissata della memoria
Gestito mediante:

- stack pointer, il registro `$sp`, ultima parola dello stack,
- frame pointer, il registro `$fp`, prima parola dello stack.

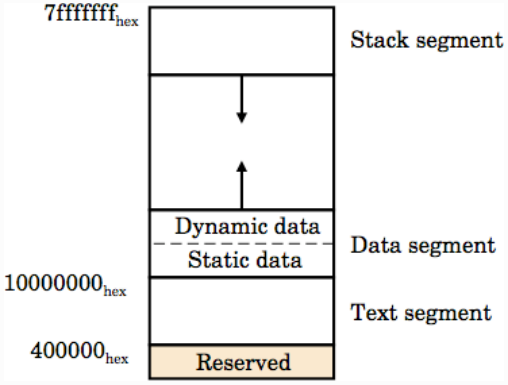
Istruzioni iniziali di una procedura: allocazione dello stack, salvataggio registri,
Istruzioni finali di una procedura: ripristino condizioni iniziali.

Esempio:

Se `fatt` deve modificare i registri `$s0`,`$ra`:

```
fatt:  addi $sp, $sp, -8  # allargo stack
       sw $s0, ($sp)
       sw $ra, 4($sp)     # salvo registri
       ...
       lw $s0, ($sp)
       lw $ra, 4($sp)     # ripristino registri
       addi $sp, $sp, 8   # ripristino stack
       jr $ra
```


Uso della Memoria



Uso della Memoria

MARS e SPIM dividono spazio indirizzamento in 5

- $0 - 400000_{hex}$: riservata al sistema operativo.
- $400000_{hex} - 10000000_{hex}$: codice programma
- $10000000_{hex} - xx$: costanti, variabili globali `$gp` (global pointer) facilita accesso
- $xx - yy$: spazio dati dinamici, syscall `sbrk n` alloca `n` byte
- $zz - 7fffffff_{hex}$: stack per chiamate di procedura

Esercizi

- Scrivere una funzione che determini se un numero è divisibile per 2 o per 3. (argomento in `$a0`, risultato in `$v0`)
- Scrivere una procedura che, in un vettore, azzeri tutti i valori divisibili per 2 o per 3.
- Scrivere una procedura che inserisca, in un vettore, di dimensione n, i primi n numeri naturali.
- Scrivere una procedura che, elimini tutti gli elementi uguali a 0 contenuti in un vettore.
- Scrivere una procedura che implementi i primi due passi del crivello di Eratostene.
- Implementare il crivello di Eratostene.

Chiamate al sistema operativo

system call

`syscall`

Simile alle chiamate di procedura. Salta ad un istruzione, fissa, del sistema operativo.

Distinta della chiamata di procedure per implementare meccanismi di protezione:
Il processore possiede due modalità di funzionamento.

- kernel: esegue qualsiasi operazione,
- utente: alcune operazione non lecite per implementare meccanismi di protezione.

MARS (SPIM) system calls

MARS e SPIM: forniscono un mini sistema operativo con una serie funzionalità base.
Registri usati per fornire parametri:

- `$v0` specifica il compito da eseguire;
- `$a0, $a1` forniscono gli argomenti;
- `$v0` può contenere il risultato finale

MARS (SPIM) system calls

Un numero limitato di possibili chiamate.
Le più utili:

operazione	cod. `$v0`	argomento	risultato
`print_int`	1	`$a0`	
`print_string`	4	base `$a0`	
`read_int`	5		`$v0`
`read_string`	8	base `$a0` lungh. `$a1`	
`exit`	10		

Esempi

Stampa il valore 32:

```
li $a0, 32
li $v0, 1
syscall
```

Stampa la lista di caratteri contenuta in memoria a partire dall'indirizzo `$s0`, fine lista marcata dal volere 0.

```
move $a0, $s0
li $v0, 4
syscall
```


Esempi

Legge 12 caratteri da terminale, da inserire in memoria a partire dall'indirizzo `$s0`

```
move $a0, $s0
li $a1, 12
li $v0, 8
syscall
```

Termina l'esecuzione del programma.

```
li $v0, 10
syscall
```


Operazioni aritmetiche - interi

Due tipi di numeri:

- interi o signed — in complemento a due;
- naturali o unsigned — (gli indirizzi considerati unsigned).

Due versioni per le operazioni aritmetiche

- per signed: `add, addi, mult ...`
- per unsigned: `addu, addiu, multu ...`

Anche: `lb, lbu`

Operazioni aritmetiche - floating-point

Terzo tipo di numeri: floating-point

- Operazioni distinte da quelle sugli interi: `add.s, add.d, sub.s, sub.d, mov.s, mov.d ...`
- Usano un diverso insieme di registri: `$fp0, ...$fp31`,
- idealmente un coprocessore, istruzioni load (store) word coprocessor, per spostare word da `$fp0, ...$fp31` a `$0, ...$31` (e viceversa),
- numeri floating-point a 32 o 64 bit: precisione singola (`add.s`) e precisione doppia (`add.d`).

Operazioni logiche

and bit a bit tra due parole,

- `and $s0, $t0, $t1`
- `andi $s0, $t0, cost`

`cost` una costante di 16 bit, viene estesa con `0`.
con pseudo-istruzioni posso inserire costanti da 32 bit.

Altre operazioni logiche:

`nor, not, or, ori, xor, xori`

Esercizi:

- `not`: pseudo-istruzione, sostituibile da ?
- calcolare il resto della divisione per 16

Shift

Spostamento bit, all'interno di una parola

shift left logical

`sll $s0, $t0, shamt`

sposta a sinistra, di `shamt` (shift-amount) posizioni, la sequenza in `$t0`, risultato in `$s0`.

shift left logical variable

`sllv $s0, $t0, $t1`

shift right logical

`srl $s0, $t0, shamt`

shift right arithmetic

`sra $s0, $t0, shamt`

Rotate

Rotazione di bit, all'interno di una parola, pseudo istruzioni,

rotate left

`rol $s0, $t0, shamt`

rotate right

`ror $s0, $t0, shamt`

Esercizi, scrivere funzioni per calcolare:

- la rotazione di una parola (senza usare la pseudo-istruzione rotate);
- il bit di parità di una parola;
- il logaritmo in base 2 di un numero.

Operazioni di confronto

- set less than: `slt rd rs rt`
- set less than unsigned: `sltu rd rs rt`
- set less than immediate: `slti rd rs imm`
-
- set less than immediate unsigned: `sltiu rd rs imm`
- ...

Esercizio, implementare le pseudo-istruzioni: set greater than equal, set greater than, set equal ?

Esercizi

1. Scrivere una procedura che determini se il processore funziona in modalità big-endian o little-endian.
2. Scrivere una procedura che determini se la stringa contenuta in memoria all'indirizzo base `$a0` e di lunghezza `$a1`, è palindroma.
3. Scrivere una procedura che determini se il contenuto del registro `$a0` è una sequenza binaria palindroma.

Strutture dati: matrici

Le matrici vengono linearizzate: ricondotte a vettori.

$$\begin{pmatrix} 1 & 3 & 5 & 7 \\ 2 & 5 & 8 & 1 \\ 3 & 7 & 1 & 5 \end{pmatrix}$$

Per riga:

1 3 5 7 2 5 8 1 3 7 1 5

o per colonna:

1 2 3 3 5 7 5 8 1 7 1 5

Matrici

Data M una matrice con n righe ed m colonne, indichiamo gli elementi con $M[i,j]$, ($i \in [0, n-1]$ e $j \in [0, m-1]$).
Nella linearizzazione di M, l'elemento $M[i,j]$ occupa la posizione:

- memorizzazione per righe: $i \times m + j$
- memorizzazione per colonne: $i + j \times n$

Nella memorizzazione per righe, se l'elemento $M[i,j]$ occupa l'indirizzo x,

- l'elemento $M[i+1,j]$ occupa l'indirizzo $x + d$,
- l'elemento $M[i,j+1]$ occupa l'indirizzo $x + d \times m$

Esercizi

1) Scrivere una funzione che ricevuti in `$a0`, `$a1` indirizzo base e lunghezza di un vettore di interi (word), calcoli, in `$v0` la somma degli elementi.

2) Scrivere una funzione che, ricevuti in `$a0` l'indirizzo base di una matrice, in `$a1` il suo numero di righe in `$a2` il numero di colonne e ricevuto in `$a3` l'indirizzo base di un vettore, inserisca nel vettore $a3 le somme degli elementi delle righe (delle colone) del vettore.

Strutture dati dinamiche

Liste (esempio tipico):

- Elemento ⇒ coppia elemento, puntatore elemento successivo.
- Puntatore ⇒ indirizzo di memoria, unsigned word

Puntatori permettono la definizione di vari strutture dati dinamiche.

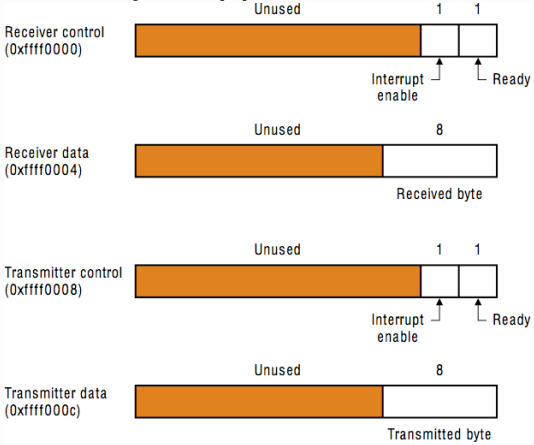
Elementi creati con una chiamata di sistema `syscall` sbrk:
alloca uno spazio di memoria libero (nel heap), parametri: `$v0` = 9, `$a0` = numero di byte da allocare, restituisce in `$v0` l'indirizzo spazio di memoria.

Esercizi

1) Scrivere una procedura che data un lista di interi, la modifica eliminando gli elementi pari e duplicando gli elementi dispari. Si richiede inoltre che il primo elemento della lista resti inalterato.

MIPS I/O

Memory mapped I/O.



MIPS Interrupt an exeption

Registri per la gestione interrupts-traps:

- BadVAddress
- Status: modalità K/U, maschera per l'interrupts (8 livelli: 5 inter. , 3 trap)
- Cause: interrupt ancora da servire,
- EPC: indirizzo dell'istruzione che ha causo l'eccezione.

Trap e interrupts forzano il salto all'indirizzo 80000080_{hex}, in kernel mode.

MIPS Interrupt an exeption

Istruzioni:

- `rfe` (return from exeption), ripristina il registro di stato
- `syscall` (system call), chiamata al sistema operativo, si passa nella modalità kernel
- `break` causa un eccezione.