Outline

•MIPS introduction with simple examples

MIPS

•**MIPS** (**Microprocessor without Interlocked Pipeline Stages**)

is a RISC microprocessor architecture developed by MIPS Technologies.

•R2000 was the first commercial MIPS CPU used in DECstation 2100 & SGI

•MIPS designs are currently primarily used in many embedded systems

MIPS

• Cisco Router (MIPS R4600)

• Laser Printers

MIPS

•MIPS CPU are available in soft-IP Cores (Synthesizable HDL)

•MIPS cores are usually found fabricated by other companies (NEC, Toshiba, IDT, etc.)

MIPS

•General-purpose registers ISA

•Load/Store ISA

operands of arithmetic instructions must be in registers

•Register size is 32 bits

•In MIPS, 32 bits are called a *word*

•Uses two characters following a dollar sign to represent a register

$s0, $s1, ..: registers that correspond to variables in high-level language program

$t0, $t1, ..: temporary registers used to hold any intermediate results

Will see some more notations and special purpose registers

MIPS

•General-purpose registers ISA

Name	Number	Use	Callee must preserve?
$zero	$0	constant 0	N/A
$at	$1	assembler temporary	no
$v0–$v1	$2–$3	Values for function returns and expression evaluation	no
$a0–$a3	$4–$7	function arguments	no
$t0–$t7	$8–$15	temporaries	no
$s0–$s7	$16–$23	saved temporaries	yes
$t8–$t9	$24–$25	temporaries	no
$k0–$k1	$26–$27	reserved for OS kernel	no
$gp	$28	global pointer	yes
$sp	$29	stack pointer	yes
$fp	$30	frame pointer	yes
$ra	$31	return address	N/A

C Assignment using Registers

C Assignment statement f = (g + h) – (i +j);

Compiler associates program variables with registers

Variables f, g, h, i, and j can be assigned to registers $s0, $s1, $s2, $s3, and $s4

What is the compiled MIPS assembly code?

Operation with 3 operands

Comments

```
add $t0, $s1, $s2        # register $t0 contains g+h
add $t1, $s3, $s4        # register $t1 contains I+j
sub $s0, $t0, $t1        # f gets $t0 - $t1
```


Complex Structures: Arrays

•Contains a number of elements instead of a single element

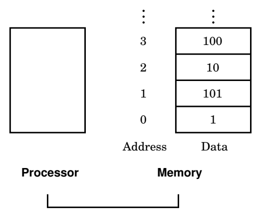
•Can be 1-dimensional or 2-dimensional

•Arrays are kept in memory due to register size limitations. Array stored starting at *base address*

•Arithmetic operation on an array element

load array element into register

•Data transfer instructions → access a word in memory → Need to supply memory address



Address of third data element is 2

Contents of Memory[2] is 10

Data from Memory to Register

Data transfer instruction: load

Format **lw** register to be loaded, constant(register used to access memory)

Memory address formed by adding constant portion and contents of second register

Example lw $t0, 8($s3) # $t0 ← Mem[8+$s3]

Compiling an assignment when an operand is in memory

Assume A is an array of 100 words.

Compiler associated variables g and h with registers $s1 and $s2

Base address of array is in $s3. The statement g = h + A[8]; MIPS assembly code?

```
lw $t0, 32($s3)        # $t0 ← A[8], Why constant is 32?
add $s1, $s2, $t0      # g = h + A[8]
```

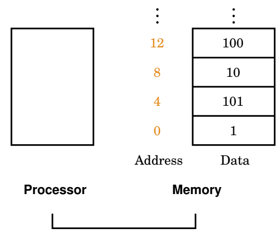

Byte Addressing

Most architectures address individual bytes

Address of a word matches address of one of the 4 bytes within the word

Words start at addresses that are multiple of 4 in MIPS (alignment restriction)

MIPS uses Big Endian (address of leftmost byte is word address)



Actual MIPS addresses are shown in figure

Byte address of third word is 8

A[8] is 9th element in array, with each element 4 bytes

Leftmost byte of A[8] is located 32 bytes away from base address (8 *4)

lw $t0, 32($s3)

Addressing Objects: Endianess and Alignment

- Big Endian address of most significant byte = word address (xx00 = Big End of word)
  - IBM 360/370, Motorola 68k, MIPS, Sparc
- Little Endian address of least significant byte = word address (xx00 = Little End of word)
  - Intel 80x86, DEC Vax, DEC Alpha (Windows NT)

msb **lsb**

$A = 12345678_H$

Addr	
0000	12
0001	34
0002	56
0003	78

big endian

	Addr
78	0000
56	0001
34	0002
12	0003

little endian

Data from Register to Memory

Data transfer instruction: store

Format **sw** register to be stored, offset(base register)

Memory address formed by adding offset and contents of base register

Example sw $t0, 48($s3) # $t0 → Mem[48+$s3]

Compiling using Load and Store

Assume A is an array of 100 words.

Compiler associated variables h with register $s2

Base address of array is in $s3. The statement A[12] = h + A[8]; MIPS assembly code?

```
lw $t0, 32($s3)        # $t0 ← A[8]
add $t0, $s2, $t0      # $t0 ← h + A[8]
sw $t0,48($s3)         # $t0 → Mem[48+$s3]
```

Another example: Variable Array Index

Assume A is an array of 100 words.

Compiler associated variables g, h, and i with register $s1, $s2, and $s4

Base address of array is in $s3. The statement g = h + A[i]; MIPS assembly code?

Need to load A[i] into register, need its address

To get address, need to multiply i by 4 to get offset of element i from base address (byte addressing issues)

4* i = 2i + 2i

```
add $t1, $s4, $s4        # $t1 ← 2*i
add $t1, $t1, $t1        # $t1 ← 4*i
add $t1, $t1, $s3        # $t1 ← base + offset
lw $t0,0($t1)            # $t0 ← Mem[0+$t1]
add $s1, $s2, $t0        # g = h + A[i]
```


Spilling Registers

- Programs have more variables than machines have registers
- Compiler tries to
  - ➢keep most frequently used variables in registers
  - ➢place rest in memory
  - ➢use loads and stores to move variables between registers and memory
- Spilling variables

  process of putting less commonly used variables (or those needed later) into memory