In the name of Allah

Midterm Overview

Zohre Soorani
Mahdi Haghverdi
Hussein Husseini
Hosna Rajaei

Isfahan University

November 18, 2023

Content

Computer Abstraction and Technology

Performace Summary

$$\text{CPU Time} = \frac{\text{CPU Clock Cycles} \times \text{CPI}}{\text{Clock Rate}}$$

$$\text{CPU Time} = \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock Cycles}}{\text{Instructions}} \times \frac{\text{Seconds}}{\text{Clock Cycles}}$$

Operations of the Computer Hardware

Operations of the Computer Hardware

Figure: Arithmatic Instructions in MIPS

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants

Example - Compiling a C Assignment Using Registers

It is the compiler's job to associate program variables with registers.
Take, for instance, the assignment statement from our earlier example:

```
f = (g + h) - (i + j);
```

The variables `f`, `g`, `h`, `i`, and `j` are assigned to the registers `$s0`, `$s1`, `$s2`, `$s3`, and `$s4`, respectively.
What is the compiled MIPS code?

Answer

- `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, which is (g + h) - (i + j)`

Operations of the Computer Hardware (Cont'd)

load word	`lw $s1, 20($s2)`	`$s1 = Memory[$s2 + 20]`	Word from memory to register
store word	`sw $s1, 20($s2)`	`Memory[$s2 + 20] = $s1`	Word from register to memory
load byte	`lb $s1, 20($s2)`	`$s1 = Memory[$s2 + 20]`	Byte from memory to register
load byte unsigned	`lbu $s1, 20($s2)`	`$s1 = Memory[$s2 + 20]`	Byte from memory to register
store byte	`sb $s1, 20($s2)`	`Memory[$s2 + 20] = $s1`	Byte from register to memory
load upper immed	`lui $s1, 20`	`$s1 = 20 *` 2^{16}	Loads constant in upper 16 bits

Table: Data Transfer Instructions in MIPS

Example - Compiling Using Load and Store

Assume variable `h` is associated with register `$s2` and the base address of the array `A` is in `$s3`.
What is the MIPS assembly code for the C assignment state ment below?

```
A[12] = h + A[8];
```

Answer

- `lw $t0, 32($s3) # Temporary reg $t0 gets A[8]`
- `add $t0, $s2, $t0 # Temporary reg $t0 gets h + A[8]`
- `sw $t0, 48($s3) # Stores h + A[8] back into A[12]`

Operations of the Computer Hardware (Cont'd)

Figure: Logical Instructions in MIPS

Logical	and	`and $s1,$s2,$s3`	`$s1` = `$s2` & `$s3`	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	`$s1` = `$s2` | `$s3`	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	`$s1` = ~ (`$s2` | `$s3`)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	`$s1` = `$s2` & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	`$s1` = `$s2` | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	`$s1` = `$s2` << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	`$s1` = `$s2` >> 10	Shift right by constant

Operations of the Computer Hardware (Cont'd)

Figure: Conditional Branch Instructions in MIPS

Conditional branch	branch on equal	`beq $s1,$s2,25`	if (`$s1` == `$s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if (`$s1`!= `$s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if (`$s2` < `$s3`) `$s1` = 1; else `$s1` = 0	Compare less than; for `beq`, `bne`
	set on less than unsigned	`sltu $s1,$s2,$s3`	if (`$s2` < `$s3`) `$s1` = 1; else `$s1` = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if (`$s2` < 20) `$s1` = 1; else `$s1` = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if (`$s2` < 20) `$s1` = 1; else `$s1` = 0	Compare less than constant unsigned

Operations of the Computer Hardware (Cont'd)

Figure: Unconditional Jump Instructions in MIPS

Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

Representing Instructions

Instructions Big Picture

Name	Format	Example						Comments
Field Size		6 bit	5 bit	5 bit	5 bit	5 bit	6 bit	All MIPS instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
`add`	R	0	18	19	17	0	32	add `$s1,$s2,$s3`
I-format	I	op	rs	rt	address			Data transfer format
`lw`	I	35	18	17	100			lw `$s1,100($s2)`
J-format	J	op	address					Unconditional Branch
`j`	J	8	300					jump to address

Logical Operations

Logical Operations Big Picture

Logical operations	C operators	Java operators	MIPS instructions
Shift left	<<	<<	`sll`
Shift right	>>	>>>	`srl`
Bit-by-bit AND	&	&	`and`, `andi`
Bit-by-bit OR	|	|	`or`, `ori`
Bit-by-bit NOT	~	~	`nor`

Instuctions for Making Decisions

Instructions for Making Decisions

- `beq register1, register2, L1`
  This instruction means go to the statement labeled `L1` if the value in `register1` equals the value in `register2`.
  The mnemonic beq stands for *branch if equal*.

- `bne register1, register2, L1`
  It means go to the statement labeled `L1` if the value in `register1` does not equal the value in `register2`.
  The mnemonic `bne` stands for *branch if not equal*.

Example - Compiling *if-then-else* into Conditional Branches

In the following code segment, `f`, `g`, `h`, `i`, and `j` are variables.
If the five variables, `f` through `j`, correspond to the five registers `$s0` through `$s4`, what is the compiled MIPS code for this C if statement?

```
if i == j:
    f = g + h
else:
    f = g - h
```

Answer

```
bne $s3, $s4, Else        # go to Else if i != j
add $s0, $s1, $s2         # f = g + h (skipped if i != j)
j   Exit                  # go to Exit
Else:
    sub $s0, $s1, $s2     # f = g - h (skipped if i = j)
Exit:
```

Example - Compiling a *while* Loop in C

Here is a traditional loop in C:

```c
while (save[i] == k){
    i += 1;
}
```

Assume that `i` and `k` correspond to registers `$s3` and `$s5` and the base of the array `save` is in `$s6`.

What is the MIPS assembly code corresponding to this C segment?

Answer

```
Loop:   sll     $t1,$s3,2         # Temp reg $t1 = i * 4
        add     $t1,$t1,$s6       # $t1 = address of save[i]
        lw      $t0,0($t1)        # Temp reg $t0 = save[i]
        bne     $t0,$s5, Exit     # go to Exit if save[i] != k
        addi    $s3,$s3,1         # i = i + 1
        j       Loop              # go to Loop
Exit:
```

Suporting Precedures in Computer Hardware

Six Steps

1. Put parameters in a place where the procedure can access them.
2. Transfer control to the procedure.
3. Acquire the storage resources needed for the procedure.
4. Perform the desired task.
5. Put the result value in a place where the calling program can access it.
6. Return control to the point of origin, since a procedure can be called from several points in a program.

Provided Registers

MIPS software follows the following convention for procedure calling in allocating its 32 registers

- `$a0-$a3`: four argument registers in which to pass parameters
- `$v0-$v1`: two value registers in which to return values
- `$ra`: one return address register to return to the point of origin

Provided Registers (Cont'd)

- In addition to allocating these registers, MIPS assembly language includes an instruction just for the procedures:
- *It jumps to an address and simultaneously saves the address of the following instruction in register* ***`$ra`****.*
- The *jump-and-link* instruction (`jal`) is simply written:
- `jal ProcedureAddress`

Provided Registers (Cont'd)

- The *link* portion of the name means that an address or link is formed that points to the calling site to allow the procedure to return to the proper address. This "link", stored in register \$ra (register 31), is called the return address.
- The return address is needed because the same procedure could be called from several parts of the program.
- To support such situations, computers like MIPS use jump register instruction (`jr`), introduced above to help with case statements, meaning an unconditional jump to the address specified in a register:
- `jr $ra`

Example - Compiling a C Procedure That Doesn't Call Another Procedure

```c
int leaf_example (int g, int h, int i, int j)
{
    int f;
    f = (g + h) - (i + j);
    return f;
}
```

Answer

```
leaf_example:
    addi $sp, $sp, -12   # adjust stack to make room for 3 items
    sw   $t1, 8($sp)     # save register $t1 for use afterwards
    sw   $t0, 4($sp)     # save register $t0 for use afterwards
    sw   $s0, 0($sp)     # save register $s0 for use afterwards

    add  $t0, $a0, $a1   # register $t0 contains g + h
    add  $t1, $a2, $a3   # register $t1 contains i + j
    sub  $s0, $t0, $t1   # f = $t0 - $t1, which is (g + h-)(i + j)

    add  $v0, $s0, $zero # returns f ($v0 = $s0 + 0)
```

Answer (Cont'd)

```
lw   $s0, 0($sp)     # restore register $s0 for caller
lw   $t0, 4($sp)     # restore register $t0 for caller
lw   $t1, 8($sp)     # restore register $t1 for caller
addi $sp, $sp, 12    # adjust stack to delete 3 items

jr   $ra             # jump back to calling routine
```

Example - Compiling a Recursive C Procedure, Showing Nested Procedure Linking

```c
int fact (int n)
{
    if (n < 1) {
        return 1;
    }
    else {
        return (n * fact(n - 1));
    }
}
```

Answer

```
fact:
    addi $sp, $sp, -8    # adjust stack for 2 items
    sw   $ra, 4($sp)     # save the return address
    sw   $a0, 0($sp)     # save the argument n

    slti $t0, $a0, 1     # test for n < 1
    beq  $t0, $zero, L1  # if n >= 1, go to L1

    addi $v0, $zero, 1   # return 1
    addi $sp, $sp, 8     # pop 2 items off stack
    jr $ra               # return to caller
```

Answer (Cont'd)

```
L1:
    addi $a0, $a0, -1  # n >= 1: argument gets (n - 1)
    jal fact           # call fact with (n - 1)

    lw $a0, 0($sp)     # return from jal: restore argument n
    lw $ra, 4($sp)     # restore the return address
    addi $sp, $sp, 8   # adjust stack pointer to pop 2 items

    mul $v0, $a0, $v0  # return n * fact (n - 1)

    jr $ra             # return to the caller
```

MIPS Addressing for 32-Bit Immediates and Addresses

Example - Loading a 32-Bit Constant

What is the MIPS assembly code to load this 32-bit constant into register `$s0`?

`0000 0000 0011 1101 0000 1001 0000 0000`

Answer

```
lui $s0, 61  # 61 decimal = 0000 0000 0011 1101 binary

# value of $s0
# 0000 0000 0011 1101 0000 0000 0000 0000

ori $s0, $s0, 2304 # 2304 decimal = 0000 1001 0000 0000

# final value of #s0
# 0000 0000 0011 1101 0000 1001 0000 0000
```

Illustration of the five MIPS addressing modes

- Immediate addressing
  where the operand is a constant within the instruc- tion itself

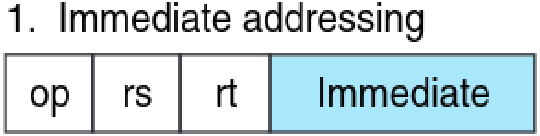


Illustration of the five MIPS addressing modes (Cont'd)

- Register addressing
  where the operand is a register

Illustration of the five MIPS addressing modes (Cont'd)

- Base or displacement addressing
  where the operand is at the memory location whose address is the sum of a register and a constant in the instruction

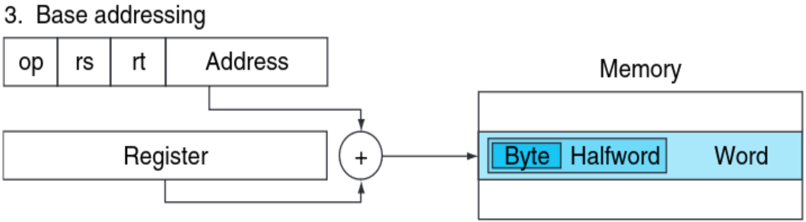


Illustration of the five MIPS addressing modes (Cont'd)

- PC-relative addressing
  where the branch address is the sum of the PC and a constant in the instruction

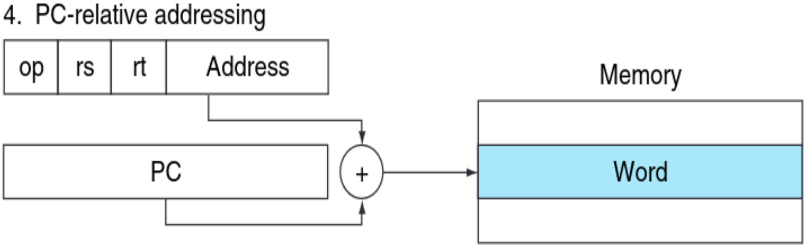
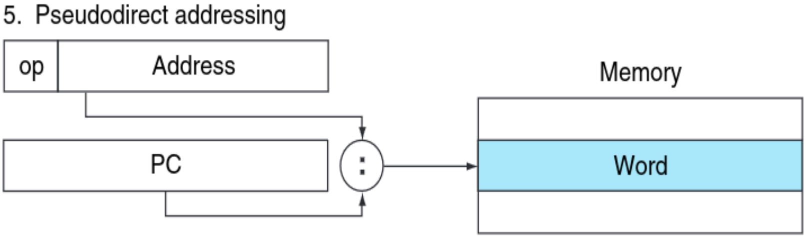


Illustration of the five MIPS addressing modes (Cont'd)

- Pseudodirect addressing
  where the jump address is the 26 bits of the instruction concatenated with the upper bits of the PC



A C Sort Example to Put It All Together

A C procedure that swaps two locations in memory

```c
void swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

What to do?

1. Allocate registers to program variables.
2. Produce code for the body of the procedure.
3. Preserve registers across the procedure invocation.

Register Allocation for `swap`

- Since swap has just two parameters, `v` and `k`, they will be found in registers `$a0` and `$a1`.
- The only other variable is `temp`, which we associate with register `$t0`

Code for the Body of the Procedure `swap`

```
swap:
    sll $t1, $a1, 2      # reg $t1 = k * 4
    add $t1, $a0, $t1    # reg $t1 = v + (k * 4)
                         # reg $t1 has the address of v[k]

    lw $t0, 0($t1)       # reg $t0 (temp) = v[k]
    lw $t2, 4($t1)       # reg $t2 = v[k + 1]
                         # refers to next element of v

    sw $t2, 0($t1)       # v[k] = reg $t2
    sw $t0, 4($t1)       # v[k+1] = reg $t0 (temp)

    jr $ra               # return to calling routine
```

A C procedure that performs a sort on the array v

```
void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i += 1) {
        for (
            j = i - 1;
            j >= 0 && v[j] > v[j + 1];
            j = 1
        ) {
            swap(v,j);
        }
    }
}
```

Register Allocation for `sort`

- The two parameters of the procedure `sort`, `v` and `n`, are in the parameter registers `$a0` and `$a1`,
- and we assign register `$s0` to `i` and register `$s1` to `j`

Code for the Body of the Procedure `sort`

Saving registers

```
sort:
    addi $sp, $sp, -20   # make room on stack for 5 registers
    sw   $ra, 16($sp)    # save $ra on stack
    sw   $s3, 12($sp)    # save $s3 on stack
    sw   $s2, 8($sp)     # save $s2 on stack
    sw   $s1, 4($sp)     # save $s1 on stack
    sw   $s0, 0($sp)     # save $s0 on stack
    ...
```

Code for the Body of the Procedure `sort` (Cont'd)

Move parameters

```
...
move $s2, $a0       # copy parameter $a0 into $s2 (save $a0)
move $s3, $a1       # copy parameter $a1 into $s3 (save $a1)
...
```

Code for the Body of the Procedure `sort` (Cont'd)

Outer loop

```
...
    move $s0, $zero          # i = 0
for1tst:
    slt  $t0, $s0, $s3       # reg $t0 = 0 if $s0 Š $s3 (i Š n)
    beq  $t0, $zero, exit1   # go to exit1 if $s0 Š $s3 (i Š n)
...
```

Code for the Body of the Procedure `sort` (Cont'd)

Inner loop

```
...
    addi $s1, $s0, -1          # j = i - 1
for2tst:
    slti $t0, $s1, 0           # reg $t0 = 1 if $s1 < 0 (j < 0)
    bne  $t0, $zero, exit2     # go to exit2 if $s1 < 0 (j < 0)
    sll  $t1, $s1, 2           # reg $t1 = j * 4
    add  $t2, $s2, $t1         # reg $t2 = v + (j * 4)
    lw   $t3, 0($t2)           # reg $t3 = v[j]
    lw   $t4, 4($t2)           # reg $t4 = v[j + 1]
    slt  $t0, $t4, $t3         # reg $t0 = 0 if $t4 Š $t3
    beq  $t0, $zero, exit2     # go to exit2 if $t4 Š $t3
...
```

Code for the Body of the Procedure `sort` (Cont'd)

Pass parameters and call `swap`

```
...
    move $a0, $s2  # 1st parameter of swap is v (old $a0)
    move $a1, $s1  # 2nd parameter of swap is j
    jal  swap        # swap
...
```

Code for the Body of the Procedure `sort` (Cont'd)

Inner loop

```
...
    addi $s1, $s1, -1   # j -= 1
    j    for2tst        # jump to test of inner loop
...
```

Code for the Body of the Procedure sort (Cont'd)

Outer loop

```
...
exit2:
    addi $s0, $s0, 1   # i += 1
    j    for1tst       # jump to test of outer loop
...
```

Code for the Body of the Procedure `sort` (Cont'd)

Restoring Registers

```
...
exit1:
    lw   $s0, 0($sp)   # restore $s0 from stack
    lw   $s1, 4($sp)   # restore $s1 from stack
    lw   $s2, 8($sp)   # restore $s2 from stack
    lw   $s3,12($sp)   # restore $s3 from stack
    lw   $ra,16($sp)   # restore $ra from stack
    addi $sp,$sp, 20   # restore stack pointer
...
```

Code for the Body of the Procedure `sort` (Cont'd)

Procedure return

```
...
jr $ra # return to calling routine
```

Questions

C to Assembly

```c
void shift(int a[], int n){
    int i;
    for(i=0;i!=n-1;i++) {
        a[i]=a[i+1];
    }
}
```

Answer

```
shift:
    move $t0, $0
    addi $t1, $a1, -1
loop:
    beq  $t0, $t1, done
    sll  $t2, $t0, 2
    add  $t2, $t2, $a0
    lw   $t3, 4($t2)
    sw   $t3, 0($t2)
    addi $t0,$t0,1
    j    loop
done:
    jr $ra
```

Recursive Functions

```
def fact(n):
    if n < 1:
        return 1
    return n * fact(n - 1)
```

Branches - C to Assembly

```
FACT: addi $sp, $sp, -8    # make room in stack for 2 more items
      sw   $ra, 4($sp)     # save the return address
      sw   $a0, 0($sp)     # save the argument n
      slti $t0, $a0, 1     # $t0 = 1 if $a0 < 1 else 0
      beq, $t0, $zero, L1 # if $t0 = 0, goto L1
      add  $v0, $zero, 1   # return 1
      add  $sp, $sp, 8     # pop two items from the stack
      jr   $ra             # return to the instruction after jal

L1:   addi $a0, $a0, -1    # subtract 1 from argument
      jal  FACT            # call fact(n - 1)
      lw   $a0, 0($sp)     # just returned from jal: restore n
      lw   $ra, 4($sp)     # restore the return address
      add  $sp, $sp, 8     # pop two items from the stack
      mul  $v0, $a0, $v0   # return n*fact(-n1)
      jr   $ra             # return to the caller
```

Recursive Functions

```python
def fib(n):
    if n < 3:
        return 1
    return fib(n - 1) + fib(n - 2)
```

Fibobacci Sequence

```
FIB:    addi $sp, $sp, -12
        sw   $ra, 8($sp)
        sw   $s1, 4($sp)
        sw   $a0, 0($sp)
        slti $t0, $a0, 3        # $t0: 1 if $a0 < 3 else 0
        beq  $t0, $zero, L1
        addi $v0, $zero, 1      # base case return 1
        j    EXIT

L1:     addi $a0, $a0, -1       # n = n - 1
        jal  FIB                # $v0 = fib(n - 1)
        addi $s1, $v0, $zero    # store the first returned result
        addi $a0, $a0, -1       # n = n - 1 -> n - 2
        jal  FIB                # $v0 = fib(n - 2)
        add  $v0, $v0, $s1      # fib(n - 1) + fib(n - 2)
```

Fibobacci Sequence (Cont'd)

```
EXIT: lw   $a0, 0($sp)
      lw   $s1, 4($sp)
      lw   $ra, 8($sp)
      addi $sp, $sp, 12
      jr   $ra
```