MIPS Pseudo Instructions and Functions

Philipp Koehn

~~16 March 2016~~
30 Sept 2019

Midterm review on
Wednesday (10/2)
→ come with questions

Midterm on Friday 10/4

pseudo instruction

Assembler

- Assembler convert readable instructions into machine code

  - assembly language: add $t0, $s1, $s2

  - machine code: 00000010 00110010 01000000 00100000

Assembler

- Assembler convert readable instructions into machine code

  – assembly language: add $t0, $s1, $s2

  – machine code: 00000010 00110010 01000000 00100000

- Make life easier with address labels

Address	Instruction
loop	...
	...
	j loop

Pseudo Instructions

- Some instructions would be nice to have

- For instance: load 32 bit value into register

$$\text{li \$s0, 32648278h}$$

- Requires 2 instructions

$$\text{lui \$s0, 3264h}$$
$$\text{ori \$s0, \$s0, 8278h}$$

- Pseudo instruction

  – available in assembly
  – gets compiled into 2 machine code instructions

Syntactic Sugar

- Move

$$\text{move \$t0, \$t1}$$

- Compiled into add instruction

$$\text{add \$t0, \$zero, \$t1}$$

Reserved Register

- Example: load word from arbitrary memory address

$$lw \ \$s0, \ 32648278h$$

- Memory address 32648278h has to be stored in register

Reserved Register

- Example: load word from arbitrary memory address

$$lw \ \$s0, \ 32648278h$$

- Memory address 32648278h has to be stored in register

- Solution: use reserved register $at   "assembler temporary"

```
lui $at, 3264h
ori $at, $s0, 8278h
lw $s0, 0($at)
```

Another Example

- Branch if less than

$$\texttt{blt \$t0, \$t1, address}$$

- Compiled into add instruction

$$\texttt{slt \$at, \$t0, \$t1}$$
$$\texttt{bne \$at, \$zero, address}$$

(slt = set if less than)

code example

- Compute $n! = n \times n-1 \times n-2 \times ... \times 2 \times 1$

- Iterative loop

  - initialize sum with n
  - loop through n-1, n-2, ..., 1
  - multiple sum with loop variable

issue: $0! = 1$

Implementation

- Registers
  - $a0: n (loop variable)
  - $v0: sum

Implementation

- Registers
  - $a0: n (loop variable)
  - $v0: sum

- Initialize

```
        move $v0, $a0              # initialize sum with n
```

Implementation

- Registers
  - $a0: n (loop variable)
  - $v0: sum

- Initialize

```
        move $v0, $a0              # initialize sum with n
```

- Loop setup

```
    loop:
        addi $a0, $a0, -1      # decrement n
        beq $a0, $zero, exit   # = 0? then done
        ...
        j loop
```

Implementation

- Registers
  - $a0: n (loop variable)
  - $v0: sum

- Initialize

```
        move $v0, $a0                # initialize sum with n
```

- Loop setup

```
    loop:
        addi $a0, $a0, -1       # decrement n
        beq $a0, $zero, exit    # = 0? then done
        ...
        j loop
```

- Multiplication

```
        mul $v0, $v0, $a0           # sum = sum * n
```

```
.text
    main:
        li $a0, 5                   # compute 5!
        move $v0, $a0               # initialize sum with n

    loop:
        addi $a0, $a0, -1           # decrement n
        beq $a0, $zero, exit        # = 0? then done
        mul $v0, $v0, $a0           # sum = sum * n
        j loop

    exit:
        jr $ra                      # done
```

jumps and subroutines

- MIPS instruction

  j address

- Only 26 bits available for address (6 bits of op-code)

Jump

- MIPS instruction

$$j \quad address$$

- Only 26 bits available for address (6 bits of op-code)

$\Rightarrow$ 32 bit address constructed by concatenating

- upper 4 bits from current program counter
- 26 bits as specified
- 2 bits with value "0"

- MIPS instruction

$$j\ address$$

- Only 26 bits available for address (6 bits of op-code)

⇒ 32 bit address constructed by concatenating

- upper 4 bits from current program counter
- 26 bits as specified
- 2 bits with value "0"

- Proper 32 bit addressing available with

$$jr\ \$register$$

Jump and Link: Subroutines

- MIPS instructions

        jal address
        jalr $register

- Address handling as before

- Stores return address in register $ra (31$^{\text{st}}$ register)

- Return from subroutine

        jr $ra

Register Conventions

- Arguments to subroutine: registers $a0, $a1, $a2, $a3   } additional args on stack

- Return values from subroutine: registers $v0, $v1, ~~$v2, $v3~~

- Conceptually

$$(\$v0, \$v1, \$v2, \$v3) = f(\$a0, \$a1, \$a2, \$a3)$$

Example

15

- Subroutine to add three numbers

```
main:
    li $a0, 10
    li $a1, 21
    li $a2, 33
    jal add3

add3:
    add $v0, $a0, $a1
    add $v0, $v0, $a2
    jr $ra
```

Another Example

- Subroutine for a + b - c

```
main:
    li $a0, 10
    li $a1, 21
    li $a2, 33
    jal add-and-sub

add-and-sub:
    add $a0, $a0, $a1
    move $a1, $a2
    jal my-sub        clobbers $ra !
    jr $ra

my-sub:
    sub $v0, $a0, $a1
    jr $ra
```

- What could go wrong?

Safekeeping

- Recursive calls: must keep return address $ra in safe place

Safekeeping

- Recursive calls: must keep return address $ra in safe place

- May also want to preserve other registers

- Temporary registers $t0-$t9 may be overwritten by subroutine *caller-saved*

- Saved registers $s0-$s7 must be preserved by subroutine *callee-saved*

- Recursive calls: must keep return address $ra in safe place

- May also want to preserve other registers

- Temporary registers $t0-$t9 may be overwritten by subroutine

- Saved registers $s0-$s7 must be preserved by subroutine

- Note

  - all this is by convention
  - you have to do this yourself

stack

- Recall: 6502

  - JSR stored return address on stack

  - RTS retrieved return address from stack

  - special instructions to store accumulator, status register

- MIPS: software stack

- By convention: stack pointer register $sp (29th register)

- Why not always use the stack? It's slow

- Store return address in saved register $s0

Alternate Idea

- Store return address in saved register $s0

- But: now have to preserve $s0 on stack

Store Return Address on Stack

- Decrease stack pointer

$$\text{addi } \$sp, \$sp, -4$$

  32-bit address has 4 bytes

- Decrease stack pointer

$$\text{addi } \$sp, \$sp, -4$$

  32-bit address has 4 bytes

- Store return address

$$\text{sw } \$ra, 0(\$sp)$$

  sw = store word

- Stack pointer points to last used address

- Load return address

$$\texttt{lw \$ra, 0(\$sp)}$$

  `lw` = store word

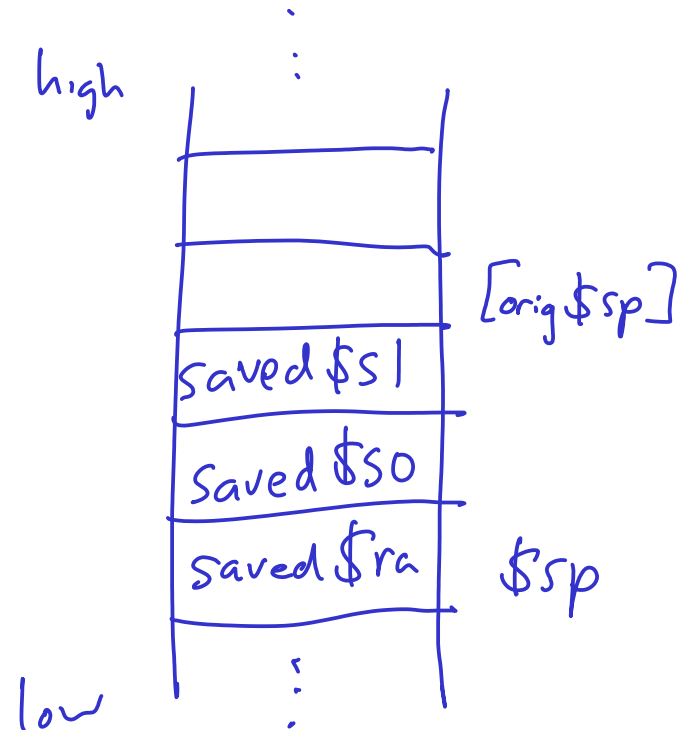- Increase stack pointer

$$\texttt{addi \$sp, \$sp, 4}$$

- Store multiple registers

```
addi $sp, $sp, -12
sw $ra 0($sp)
sw $s0 4($sp)
sw $s1 8($sp)
```

- Load

```
lw $ra 0($sp)
lw $s0 4($sp)
lw $s1 8($sp)
addi $sp, $sp, 12
```

high

[orig $sp]

saved $s1

saved $s0

saved $ra $sp

Frame Pointer

- What if we want to consult values stored on the stack?

- Example

  - subroutine stores return address and some save registers on stack
  - some code does something
    (maybe even store more things on stack)
  - subroutine wants to consult stored return address

- What if we want to consult values stored on the stack?

- Example

  – subroutine stores return address and some save registers on stack
  – some code does something
    (maybe even store more things on stack)
  – subroutine wants to consult stored return address

- Stack pointer has changed

  $\rightarrow$ may be difficult to track down

Frame Pointer

- What if we want to consult values stored on the stack?

- Example

  – subroutine stores return address and some save registers on stack
  – some code does something
    (maybe even store more things on stack)
  – subroutine wants to consult stored return address

- Stack pointer has changed

  $\rightarrow$ may be difficult to track down

- Solution

  – store entry stack pointer in frame pointer $fp (30th register)
        move $fp, $sp
  – retrieve return address using frame pointer
      * lw $s0, 0($fp)

example

```
.text
        li $a0, 5                   # compute 5!
        move $v0, $a0               # initialize sum with n

    loop:
        addi $a0, $a0, -1           # decrement n
        beq $a0, $zero, exit        # = 0? then done
        mul $v0, $v0, $a0           # sum = sum * n
        j loop

    exit:
        jr $ra                      # done
```

Implemented as a Function

- Subroutine call (function argument in $a0)

```
main:
    li $a0, 5                    # compute 5!
    jal fact                     # call function
```

- Return from subroutine (return value is in $v0)

```
exit:
    jr $ra                       # done
```

```
.text
    main:
        li $a0, 5                   # compute 5!
        jal fact                    # call function
        jr $ra                      # done


    fact:
        (old code)


    exit:
        jr $ra                      # done
```

```
.text
    main:
        li $a0, 5                      # compute 5!
        jal fact                       # call function
        jr $ra                         # done

    fact:
        move $v0, $a0                  # initialize sum with n

    loop:
        addi $a0, $a0, -1              # decrement n
        beq $a0, $zero, exit          # = 0? then done
        mul $v0, $v0, $a0             # sum = sum * n
        j loop

    exit:
        jr $ra                         # done
```

clobbers $ra!

Recursive Implementation

- Idea: f(n) = f(n-1) * n

    $$f(0) = 1 \qquad - \text{base case}$$

- Recursive call needs to preserve

    – return address

    – argument (n)

Termination Condition

- Check if argument is 0

```
fact:
      beq $a0, $zero, final      # = 0? then done


      (common case)

final:
      li $v0, 1
      jr $ra                          # done
```

- Note: no need to preserve registers

Core Recursion

- Recursive call f(n-1)

```
        addi $a0, $a0, -1          # decrement n
        jal fact                   # recursive call -> $v0 is f(n-1)
```

- Multiply with argument

```
        mul $v0, $v0, $a0          # f(n-1) * n
```

- Save registers

```
addi $sp, $sp, -8
sw $ra,0($sp)          # return address on stack
sw $a0,4($sp)          # argument on stack
```

- Restore registers

```
lw $ra,0($sp)          # return address from stack
lw $a0,4($sp)          # argument from stack
addi $sp, $sp, 8
```

Complete Code

```
fact:
        beq $a0, $zero, final      # = 0? then done

        addi $sp, $sp, -8
        sw $ra,0($sp)              # return address on stack
        sw $a0,4($sp)              # argument on stack

        addi $a0, $a0, -1          # decrement n
        jal fact                   # recursive call -> $v0 is f(n-1)

        lw $ra,0($sp)              # return address from stack
        lw $a0,4($sp)              # argument from stack
        addi $sp, $sp, 8

        mul $v0, $v0, $a0          # f(n-1) * n
        jr $ra                     # done

final:
        li $v0, 1
        jr $ra                     # done
```