ECE 250 / CS 250
Computer Architecture

Assembly Programming

Benjamin Lee

Slides based on those from Alvin Lebeck, Daniel Sorin, Andrew Hilton, Amir Roth, Gershon Kedem

Today's Lecture

Admin

- Midterm #1 – Tuesday, Feb 18
- Midterm #2 – Thursday, Apr 3

Outline

- Review
- MIPS Assembly Programming

Reading

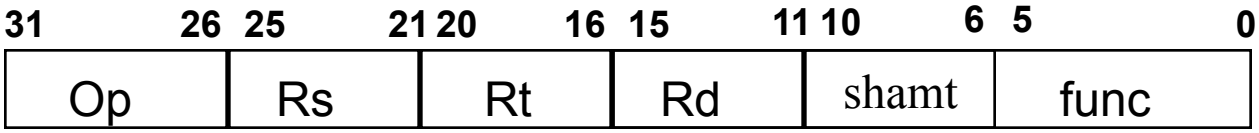
- Chapter 2 – Instructions
- An overview of SPIM and the MIPS32 instruction set @ http://spimsimulator.sourceforge.net/HP_AppA.pdf
- MIPS Quick Ref (PDF in docs section of web site)

Review: MIPS ISA Categories

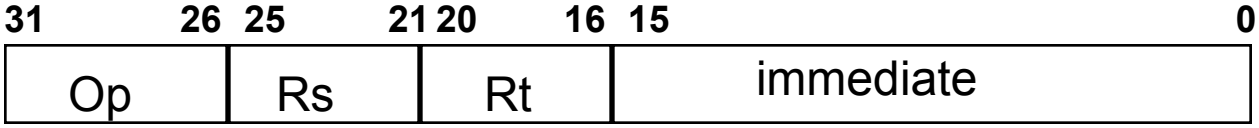
- Arithmetic
  - add, sub, mul, etc
- Logical
  - AND, OR, SHIFT
- Data Transfer
  - load, store
  - MIPS is LOAD/STORE architecture
- Conditional Branch
  - implement if, for, while… statements
- Unconditional Jump
  - support method invocation (procedure calls)

Review: MIPS Instruction Formats

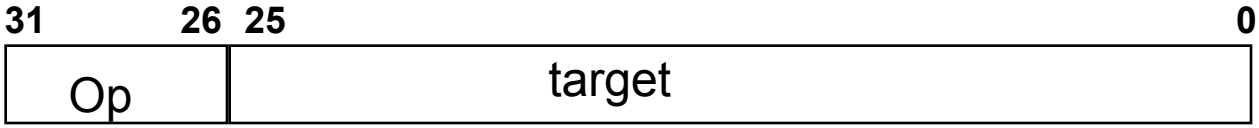
R-type: Register-Register



I-type: Register-Immediate



J-type: Jump / Call

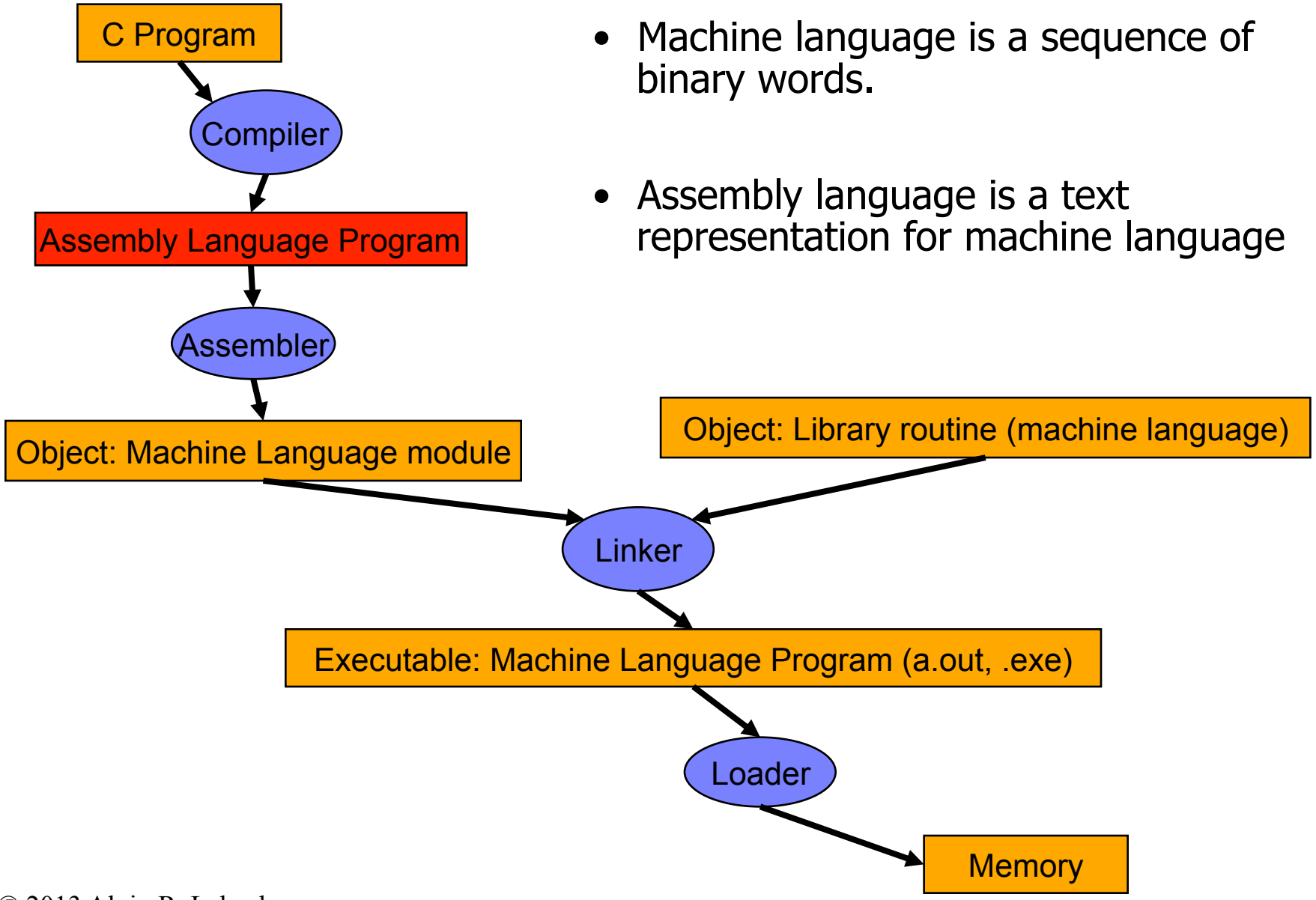


Terminology

Op = opcode

Rs, Rt, Rd = register specifier

Compilers, Linkers, Loaders



- Machine language is a sequence of binary words.
- Assembly language is a text representation for machine language

MIPS Assembly Language

- One instruction per line.
  - add $2, $3, $4 => $2 = $3+$4 (result is first operand, usually)

- Numbers: base-10 integers or Hex w/ leading 0x.
- Identifiers: alphanumeric, _, . string starting with letter or _
- Labels: identifiers starting at the beginning of a line followed by ":"
- Comments: everything following # till end-of-line.

- Instruction format: Space and "," separated fields.
  - [Label:] <op> reg1, [reg2], [reg3] [# comment]
  - [Label:] <op> reg1, offset(reg2) [# comment]
  - .Directive [arg1], [arg2], . . .

Assembly Language (cont.)

- Pseudo-instructions: extend the instruction set for convenience
- Examples
  - ```
    move $2, $4          # $2 = $4, (copy $4 to $2)
    add $2, $4, $0
    ```
  - ```
    li $8, 40            # $8 = 40, (load 40 into $8)
    addi  $8, $0, 40
    ```
  - ```
    sd $4, 0($29)        # mem[$29] = $4; Mem[$29+4] = $5
    sw $4, 0 ($29)
    sw $5, 4($29)
    ```
  - ```
    la $4, 0x1000056c    # Load address $4 = <address>
    lui $4, 0x1000
    ori $4, $4, 0x056c
    ```

Assembly Language (cont.)

- Directives: tell the assembler what to do...
- Format "."<string> [arg1], [arg2] . . .

- Examples

```
.data   [address]        # start a data segment.
                         #  [optional beginning address]
.text  [address]         # start a code segment.
.align n                 # align segment on 2^n  byte boundary.
.ascii <string>          # store a string in memory.
.asciiz <string>         # store a null terminated string in memory
.word w1, w2, . . . , wn  # store n words in memory.
```

A Simple Program

- Add two numbers x & y together

```
   .text                # declare text segment
   .align 2             # align it on 4-byte boundary
main:                   # label for main
   la   $3, x           # load address of x into R3 (pseudo-inst)
   lw   $4, 0($3)       # load value of x into R4
   la   $3, y           # load address of y into R3 (pseudo-inst)
   lw   $5, 0($3)       # load value of y into R5
   add  $6, $4, $5      # compute x+y
   jr   $31             # return to calling routine

   .data                # declare data segment
   .align 2             # align it on 4-byte boundary
x: .word 10             # initialize x to 10
y: .word 3              # initialize y to 3
```

Control Idiom: If-Then-Else

- Introduce programming idioms in assembly
  - Why? Hardware is optimized for common case

- First control idiom: if-then-else

```
if (A < B) A++;        // assume A in register $s1
else B++;              // assume B in $s2
```

```
        slt  $s3,$s1,$s2   // if $s1<$s2, then $s3=1
        beqz $s3,else      // branch to else if !condition
        addi $s1,$s1,1
        b    join          // branch to join
else:   addi $s2,$s2,1
join:
```

Assembler converts "else" target of beqz into immediate → what is the immediate?

Control Idiom: Arithmetic For Loop

- Second idiom: "for loop" with arithmetic induction

```
int A[100], sum, i, N;
for (i=0; i<N; i++){          // assume: i in $s1, N in $s2
  sum += A[i];                // &A[i] in $s3, sum in $s4
}
```

```
        sub  $s1,$s1,$s1   // initialize i to 0
loop:   slt  $t1,$s1,$s2   // if i<N, then $t1=1
        beqz $t1,exit      // test for exit at loop header
        lw   $t1,0($s3)    // $t1 = A[i]  (not &A[i])
        add  $s4,$s4,$t1   // sum = sum + A[i]
        addi $s3,$s3,4     // increment &A[i] by sizeof(int)
        addi $s1,$s1,1     // i++
        b loop             // backward branch
exit:
```

Control Idiom: Pointer For Loop

- Third idiom: for loop with pointer induction

```
struct node_t { int val; struct node_t *next; };
struct node_t *p, *head;
int sum;
for (p=head; p!=NULL; p=p->next)   // p in $s1, head in $s2
   sum += p->val                   // sum in $s3
```

```
        add $s1,$s2,$0     // p = head
loop:   beq $s1,$0,exit    // if p==0 (NULL), goto exit
        lw $t1,0($s1)      // $t1 = *p = p→val
        add $s3,$s3,$t1    // sum = sum + p→val
        lw $s1,4($s1)      // p = *(p+1) = p→next
        b loop
exit:
```

The C code

```
#include <stdio.h>

int main ( )
{
  int i;
  int sum = 0;
  for(i=0; i <= 100; i++)
       sum = sum + i*i ;
  printf ("The answer is %d\n",sum);
}
```

The Assembly Code

```
   .text
   .align       2

main:
   move $14, $0 # i = 0
   move $15, $0 # tmp = 0
   move $16, $0 # sum = 0

loop:
   mul  $15, $14, $14  # i*i
   add  $16, $16, $15  # sum+i*i
   addi $14, $14, 1    # i++
   ble  $14, 100, loop # i < 100

go print answer
exit
```

Example 2

Task: sum together the integers stored in memory

```
.text           # Code
.align  2       # align on word boundary
.globl  main    # declare main
main:           # MAIN procedure Entrance


# fill in what goes here


        .data                   # Start of data segment
list:   .word   35, 16, 42, 19, 55, 91, 24, 61, 53
msg:    .asciiz "The sum is "
nln:    .asciiz "\n"
```

Procedure Call and Return

```
int equal(int a1, int a2) {
   int tsame;
   tsame = 0;
   if (a1 == a2)
         tsame = 1;
         return(tsame);
}
main()
{
   int x,y,same;
   x = 43;
   y = 2;
   same = equal(x,y);
   // other computation
}
```

0x10000	addi $1, $0, 43
0x10004	addi $2, $0, 2
0x10008	jal 0x30408
0x1000c	??

0x30408	addi $3, $0, 0
0x3040c	bne $1, $2, 4
0x30410	addi $3, $0, 1
0x30414	jr $31

PC	$31
0x10000	??
0x10004	??
0x10008	??
0x30408	0x1000c
0x3040c	0x1000c
0x30410	0x1000c
0x30414	0x1000c
0x1000c	0x1000c

Procedure Call GAP

ISA Level

- call and return instructions

C Level

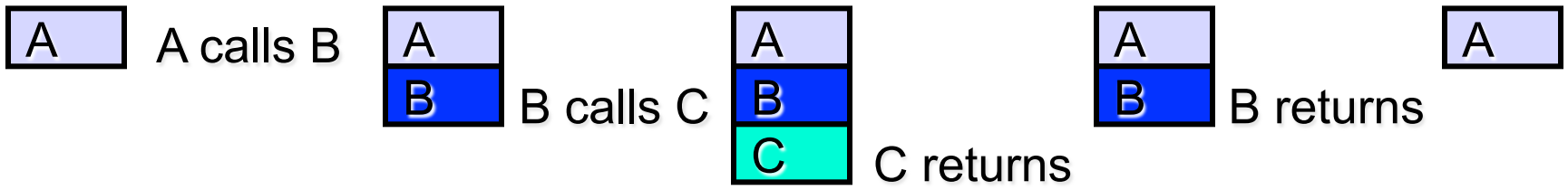
- Local Name Scope
  - change tsame to same
- Recursion
- Arguments and Return Value (functions)

Assembly Level

- Must bridge gap between HLL and ISA
- Supporting Local Names
- Passing Arguments (arbitrary number?)

Control Idiom: Procedure Call

- Procedure calls obey stack discipline
  - Local procedure state contained in stack frame
  - When a procedure is called, a new frame opens
  - When a procedure returns, the frame collapses
- Procedure stack is in memory
  - Distinct from operand stack which is not addressable
- Procedure linkage implemented by convention
  - Called procedure ("callee") expects frame to look a certain way
    - Input arguments and return address are in certain places
  - Caller "knows" this



Summary

- Assembler Translates Assembly to Machine code
- Pseudo Instructions

Next Time

- Procedure calls, MIPS calling conventions

Reading

- An overview of SPIM and the MIPS32 instruction set @ http://spimsimulator.sourceforge.net/HP_AppA.pdf
- MIPS Quick Ref (PDF in docs section of web site)