Lecture 5: MIPS Examples

- Today's topics:

  - the compilation process
  - full example – sort in C

- Reminder: 2nd assignment will be posted later today

Dealing with Characters

- Instructions are also provided to deal with byte-sized and half-word quantities: lb (load-byte), sb, lh, sh

- These data types are most useful when dealing with characters, pixel values, etc.

- C employs ASCII formats to represent characters – each character is represented with 8 bits and a string ends in the null character (corresponding to the 8-bit number 0)

Example

Convert to assembly:

```
void strcpy (char x[], char y[])
{
    int i;
    i=0;
    while  ((x[i] = y[i]) != `\0')
    i += 1;
}
```

Example

Convert to assembly:

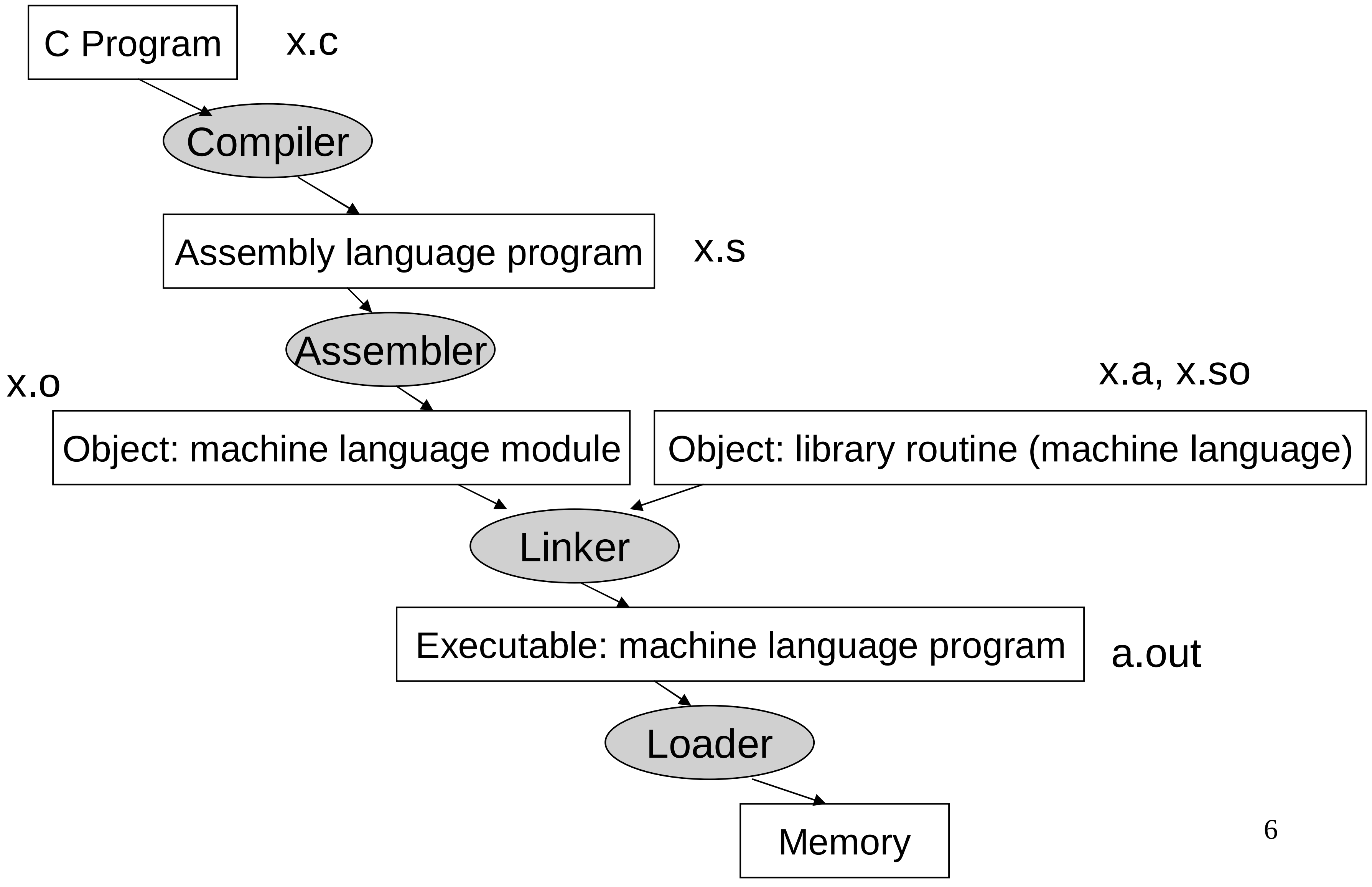
```
void strcpy (char x[], char y[])
{
    int i;
    i=0;
    while  ((x[i] = y[i]) != `\0’)
      i += 1;
}
```

```
strcpy:
 addi    $sp, $sp, -4
 sw      $s0, 0($sp)
 add     $s0, $zero, $zero
L1: add  $t1, $s0, $a1
 lb      $t2, 0($t1)
 add     $t3, $s0, $a0
 sb      $t2, 0($t3)
 beq     $t2, $zero, L2
 addi    $s0, $s0, 1
 j       L1
L2: lw   $s0, 0($sp)
 addi    $sp, $sp, 4
 jr      $ra
```

Large Constants

- Immediate instructions can only specify 16-bit constants
- The lui instruction is used to store a 16-bit constant into the upper 16 bits of a register… thus, two immediate instructions are used to specify a 32-bit constant
- The destination PC-address in a conditional branch is specified as a 16-bit constant, relative to the current PC
- A jump (j) instruction can specify a 26-bit constant; if more bits are required, the jump-register (jr) instruction is used

Starting a Program



Role of Assembler

- Convert pseudo-instructions into actual hardware instructions – pseudo-instrs make it easier to program in assembly – examples: “move”, “blt”, 32-bit immediate operands, etc.

- Convert assembly instrs into machine instrs – a separate object file (x.o) is created for each C file (x.c) – compute the actual values for instruction labels – maintain info on external references and debugging information

Role of Linker

- Stitches different object files into a single executable
  - patch internal and external references
  - determine addresses of data and instruction labels
  - organize code and data modules in memory
- Some libraries (DLLs) are dynamically linked – the executable points to dummy routines – these dummy routines call the dynamic linker-loader so they can update the executable to jump to the correct routine

Full Example – Sort in C

```
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);
        }
    }
}
```

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

- Allocate registers to program variables
- Produce code for the program body
- Preserve registers across procedure invocations

The swap Procedure

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

- Allocate registers to program variables
- Produce code for the program body
- Preserve registers across procedure invocations

The swap Procedure

- Register allocation: $a0 and $a1 for the two arguments, $t0 for the temp variable – no need for saves and restores as we're not using $s0-$s7 and this is a leaf procedure (won't need to re-use $a0 and $a1)

```
swap:   sll   $t1, $a1, 2
        add   $t1, $a0, $t1
        lw    $t0, 0($t1)
        lw    $t2, 4($t1)
        sw    $t2, 0($t1)
        sw    $t0, 4($t1)
        jr    $ra
```

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

The sort Procedure

- Register allocation: arguments v and n use $a0 and $a1, i and j use $s0 and $s1

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

The sort Procedure

- Register allocation: arguments v and n use $a0 and $a1, i and j use $s0 and $s1; must save $a0, $a1, and $ra before calling the leaf procedure

- The outer for loop looks like this: (note the use of pseudo-instrs)

```
           move   $s0, $zero        # initialize the loop
loopbody1: bge    $s0, $a1, exit1   # will eventually use slt and beq
           … body of inner loop …
           addi   $s0, $s0, 1
           j       loopbody1
exit1:
```

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

The sort Procedure

- The inner for loop looks like this:

```
            addi    $s1, $s0, -1        # initialize the loop
loopbody2:  blt     $s1, $zero, exit2   # will eventually use slt and beq
            sll     $t1,  $s1, 2
            add     $t2, $a0, $t1
            lw      $t3, 0($t2)
            lw      $t4, 4($t2)
            bge     $t4, $t3, exit2
            … body of inner loop …
            addi    $s1, $s1, -1
            j        loopbody2
exit2:
```

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

Saves and Restores

- Since we repeatedly call “swap” with \$a0 and \$a1, we begin “sort” by copying its arguments into \$s2 and \$s3 – must update the rest of the code in “sort” to use \$s2 and \$s3 instead of \$a0 and \$a1

- Must save \$ra at the start of “sort” because it will get over-written when we call “swap”

- Must also save \$s0-\$s3 so we don’t overwrite something that belongs to the procedure that called “sort”

Saves and Restores

```
sort:   addi   $sp, $sp, -20
        sw     $ra, 16($sp)
        sw     $s3, 12($sp)
        sw     $s2, 8($sp)
        sw     $s1, 4($sp)
        sw     $s0, 0($sp)
        move   $s2, $a0
        move   $s3, $a1
          …
        move   $a0, $s2      # the inner loop body starts here
        move   $a1, $s1
        jal    swap
          …
exit1:  lw     $s0, 0($sp)
          …
        addi   $sp, $sp, 20
        jr     $ra
```

9 lines of C code → 35 lines of assembly

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

Relative Performance

Gcc optimization	Relative performance	Cycles	Instruction count	CPI
none	1.00	159B	115B	1.38
O1	2.37	67B	37B	1.79
O2	2.38	67B	40B	1.66
O3	2.41	66B	45B	1.46

- A Java interpreter has relative performance of 0.12, while the Jave just-in-time compiler has relative performance of 2.13

- Note that the quicksort algorithm is about three orders of magnitude faster than the bubble sort algorithm (for 100K elements)

IA-32 Instruction Set

- Intel's IA-32 instruction set has evolved over 20 years – old features are preserved for software compatibility
- Numerous complex instructions – complicates hardware design (Complex Instruction Set Computer – CISC)
- Instructions have different sizes, operands can be in registers or memory, only 8 general-purpose registers, one of the operands is over-written
- RISC instructions are more amenable to high performance (clock speed and parallelism) – modern Intel processors convert IA-32 instructions into simpler micro-operations

Title

- Bullet