

# CS2810 Lecture Notes – Set 2

Chapters 2-3 of text  
Instruction Sets/MIPS Architecture

# MIPS Integer Multiplication

- Integers on a MIPS machine are represented as 32-bit 2's complement values.
  - Thus, when two integers are multiplied together, the result can be up to 64 bits.
- On a multiply the result does not go into a single register, but into two special purpose registers Hi and Lo.
- The only instructions that access these registers are mfhi and mflo

# Multiplication & Division (Integer)

- `mult $s2,$s3`
  - $\{hi,lo\} = \$s2 * \$s3$  i.e. the 64-bit result
- `mfhi $s2`
- `mflo $s3`
  
- `mul $t3,$t0,$t1 # $t3=$t0*$t1; pseudo`
  
- Neither of these mult's check for overflow

# Question

- If I execute `mul $t0, $t1, $t2` what will end up in `$t0`

| <code>\$t1</code> | <code>\$t2</code> | <code>\$t0</code> |
|-------------------|-------------------|-------------------|
| -2                | +2                |                   |
| 100               | 100               |                   |
| $2^{(30)}$        | $2^{(30)}$        |                   |
| $2^{(31)} - 1$    | 2                 |                   |
| $-2^{(31)}$       | 2                 |                   |
| $-2^{(31)}$       | -2                |                   |

# Question

- If I execute `mul $t0, $t1, $t2` what will end up in `$t0`

| <code>\$t1</code> | <code>\$t2</code> | <code>\$t0</code> |
|-------------------|-------------------|-------------------|
| -2                | +2                | -4                |
| 100               | 100               | 10000             |
| $2^{(30)}$        | $2^{(30)}$        | 0                 |
| $2^{(31)} - 1$    | 2                 | -2                |
| $-2^{(31)}$       | 2                 | 0                 |
| $-2^{(31)}$       | -2                | 0                 |

# MIPS Multiplication & Division

- `div $s2,$s3`
  - `lo` =  $\$s2 / \$s3$ , `hi` =  $\$s2 \bmod \$s3$ , i.e. the remainder
- `div $t0,$t1,$t2`
- `mfhi $s1`
- `mflo $s1`

# Overflow for Multiplication

- What constitutes **no** overflow on a multiply?
  - All bits of  $hi$  == sign bit of low

# Overflow for Multiplication

- Now how do we test for overflow on a multiply?
- All bits of HI = sign bit of LO
- How would you test for this condition?

# Functions and Procedures

- Why?
  - Reuse
  - Work separation
  - Debugging/understanding
- Why not?
  - Speed
  - Side effects or unexpected impact

# Time vs Space

- When you use procedures, what are the tradeoffs with simply writing the code – in line?
  - More efficient in terms of code if can reuse
  - Slower because of parameter passing, setup, calls, etc.

# Functions or Procedures

- There is really no strict convention for how to call and pass parameters for all MIPS applications, or even other architectures.
- In general, every computer will have the following
  - A convention for passing in and out parameters in registers
  - An instruction to jump to, and return from a function/procedure
  - A dynamic memory for parameters, etc.
    - system stack pointer and a frame pointer

# Ignore & Jump right in

- Set up space for passed parameters
  - Load into registers
- Jump to procedure, but remember from whence you came
  - Jal proc\_address
  - This also saves PC + 4 in \$31 or \$ra
- “unload” passed parameters if necessary
  - .....
- Allocate storage for local variables if necessary
  - .....
- Execute
- Pass results back
- Return from whence you came
  - Jr \$ra

# Procedure – An Example

```
//this example uses unnecessary steps for
// demonstration purposes
int sum( int num1, int num2, int num3, int num4)
{
    int answer, part1, part2;
    part1 = num1 + num2;
    part2 = num3 + num4;
    answer = part1 + part2;
    return answer;
}
```

# Procedures

- For this procedure we have
  - 4 passed (by value) parameters \$a0,..\$a3
  - one returned value \$v0
  - 3 local variables \$t0, ..., \$t2

# Procedures

- Remember, the steps are:
  - Set up space for passed parameters

***lw \$a0,num1***

***lw \$a1,num2***

***lw \$a2,num3***

***lw \$a3,num4***

# Procedures

- Remember, the steps are:
  - Jump to procedure, but remember from whence you came

***jal sum***

- “unload” passed parameters

***Don't need to in this case***

- Allocate storage for local variables

***Add \$t0,\$zero,\$zero***

***Add \$t1,\$zero,\$zero***

***Add \$t2,\$zero,\$zero***

- Execute

.....

# Procedures

– Pass results back

***\$add \$v0,....***

– return

jr \$ra

# Procedures - Example

- Implement an iterative procedure to calculate  $N!$ 
  - Use `$a0` to pass the value of  $N$
  - Use `$v0` to return the value of  $N!$
  - Ignore the possibility of overflow

# Factorial(N)

```
int fact(int N)
{
    if (N<2) return 1;
    else
        for (i=2;i<=N;i++)
            fact = fact*i;
return fact;
}
```

# Calling Factorial

```
lw $a0, ????? # the value to compute  
# factorial of
```

```
jal fact
```

```
# at this point, $v0 should have the factorial
```

```
fact:          bge $a0,0,OK
                li $v0,-1
                jr $ra
OK:            li    $t0,1
OVR:          ble $a0,1,DONE
                mul $t0,$t0,$a0
                addi $a0,$a0,-1
                b OVR
DONE:         move $v0,$t0
                jr   $ra
```

# Procedures

- What is the problem with the preceding techniques for parameter passing?
  - A function/procedure cannot call another
  - Why?
    - Parameters are saved in static locations
- What is the solution?
  - Use a stack

# F&P

- There is one important convention to remember – Caller vs Callee
  - Caller – is the calling program/function/procedure
  - Callee- is the called program

# F&P

- In general, what must happen for general function calling is the following
- Caller
  - Put passed parameters in a location that the callee will know and have access to
  - Jump to the callee's start address and at the same time save the PC contents (the return address) somewhere
  - On return, extract the returned values

# F&P

- Callee
  - Extract the passed parameters
  - If any non-temporary registers are used, save their contents
  - Allocate space for the local variables
  - Save return address somewhere so can call a function from this function
  - Execute
  - Restore or set-up return values/parameters
  - Restore return address and return

# F&P

- A convention for passing in/out parameters in registers
  - In MIPS, that convention is the “a” registers, i.e. \$a0-\$a3, which are actually \$4-\$7
    - These are integer or memory values, and hence they won't work for character or floating point data
  - MIPS also designates \$s0-\$s7 as saved registers
    - If the callee uses any of the \$s# registers, it must first save their contents
  - MIPS allows the return of up to two parameters in register \$v0 and \$v1
- An instruction to jump to, and return from a function/procedure
  - Jal \$ra
  - Jr \$ra

# What to save and what not to save

- MIPS uses the following convention to split the register saving chores.
- The *caller* is responsible for saving and restoring any of the following **caller-saved registers** that it cares about.

\$t0-\$t9

\$a0-\$a3

\$v0-\$v1

In other words, the callee may freely modify these registers, under the assumption that the caller already saved them if necessary.

# What to save and what not to save

- MIPS uses the following convention to split the register saving chores.
- The *callee* is responsible for saving and restoring any of the following **callee-saved registers** that it uses. (Remember that \$ra is “used” by jal.)

\$s0-\$s7

\$ra

Thus the caller may assume these registers are not changed by the callee.

- \$ra is tricky; it is saved by a callee who is also a caller.

# F&P

- Everyone's system today uses a stack for In/Out parameters, and locals, etc.
  - Allows for multiple levels of calls
  - Allows local variable allocation and deallocation
  - The MIPS system stack is \$sp
  - The SPIM and MARS emulators set \$sp to point to the highest location in memory – thus you do not need to initialize it

# F&P

- Remember – stacks have essentially 4 operations
  - Push
  - Pop
  - Full() (T/F)
  - Empty() (T/F)
- At the assembly level, we really only have Push and Pop, so beware
  - If the stack grows too large, you won't get stack overflow, you will overwrite the program

# F&P

- Push in MIPS

```
sw    $t0,($sp)
addi  $sp,$sp, -4
```

Note: you can also push a byte onto the stack, but be careful because you may then not be on a word boundary

# F&P

- Pop in MIPS

```
addi $sp,$sp,4  
lw   $t0,($sp)
```

Note: you can also pop a byte from the stack, but be careful because you may then not be on a word boundary

# F&P Calling Process

- Stack operations

Let's say four integers have been pushed onto the stack, a, b, and c

The following instructions show how to load each parameter into register \$t0

```
lw    $t0,0($t0) # loads c into $t0
```

```
lw    $t0,4($sp)# loads b into $t0
```

```
lw    $t0,8($sp)# loads a into $t0
```

# F&P Note

## One Way for Push and Pop

(push)

```
sw    $t0,($sp)
addi  $sp,$sp,-4
```

(pop)

```
addi  $sp,$sp,4
lw    $t0,($sp)
```

## Another way for push and pop

(push)

```
addi  $sp,$sp,-4
sw    $t0,($sp)
```

(pop)

```
lw    $t0,($sp)
addi  $sp,$sp,4
```

# The Frame Pointer

- With parameters pushed onto the stack, all references can be with respect to the stack pointer.
- Problem
  - The problem with this method is that in the callee, there may be local variables pushed onto the stack. Thus, the location of a variable with respect to the stack pointer will vary as local variables are added to the stack.

# The Frame Pointer

- The solution to the problem is to have a second register (MIPS calls it the frame pointer) \$fp
- The frame pointer contains a pointer (address) to the location on the stack of the end of the frame for that function. Thus passed parameters are accessed relative to the frame pointer and local variables are accessed relative to the top of stack.

# Example

- Assume that the caller has two parameters and it has pushed them onto the stack before the jal (call them P1 and P2)
- The stack on entry to the callee (function) will look like:

P2 <- (\$sp)

P1

# Example cont'd

- In the callee, the first order of business is to set up the frame. This is done by saving the current frame point and the return address on the stack

```
$fp      <-$sp  
$ra      <- $fp  
P2  
P1
```

# Example cont'd

- Now as local variables are allocated, the passed parameters are all relative to the \$fp

```
local2          <- $sp  
local1  
$fp  
$ra            <- $fp  
P2  
P1
```

# Example Cont'd

- Now – how will we return, etc.
- What about returning parameters to the calling routine?

# Recursion

- While the multiplication of two integers is a single MIPS instruction, one can also view it as a recursive function

```
Int multiply(int a, int b)  # assume a >= b >= 0
    if (b == 0) return 0;
        else return a + multiply(a, b-1);
```

# Recursion

- Do it

# Character Operations/Strings

- We are working with 4-byte words
  - characters are only 1-byte long, the assembler stores individual characters in an entire 32-bit register.
- The following command takes the right-most byte of register \$t0 and copies it to the memory specified:
  - `sb $t0, 0($s0)`
- The following command gets a single byte from memory and stores it in the right-most byte of register \$t0:
  - `lb $t0, 0($s0)`
- The following command allocates 10 bytes of storage
  - `.space 10`
- The following command stores values in consecutive bytes
  - `.byte 5,-3,10`

# Character Operations/Strings

## Cat Function

```
void strcat(char name1[], char name2[])
{ int i, j;
  i =0;   j=0;
  while(name1[i] != 0) i=i+1;
  while(name2[j] != 0)
  {
    name1[i] = name2[j];
    i=i+1;
    j=j+1;
  }
  name1[i] = name2[j]; //copy null terminating character
}
```

# Do It

- Can you implement the previous pseudo-code in MIPS?

# Character Codes

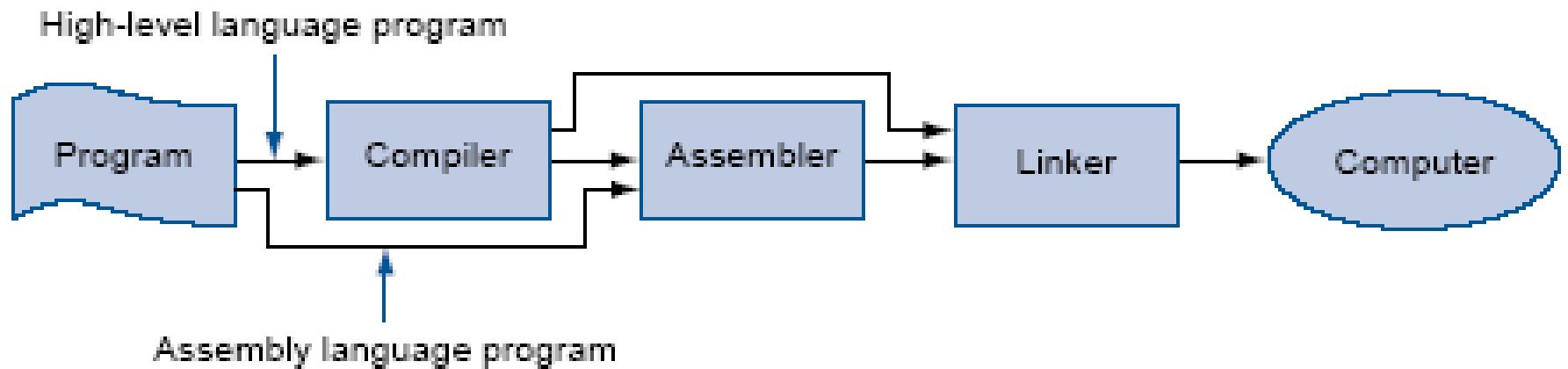
- ASCII is a 7 bit code
  - How many characters can a 7-bit code represent?
- For a more rich user interface, longer codes have been developed
  - Java uses unicode – 16 bits
  - Allows for representations for different languages, usually in blocks

# Character Codes

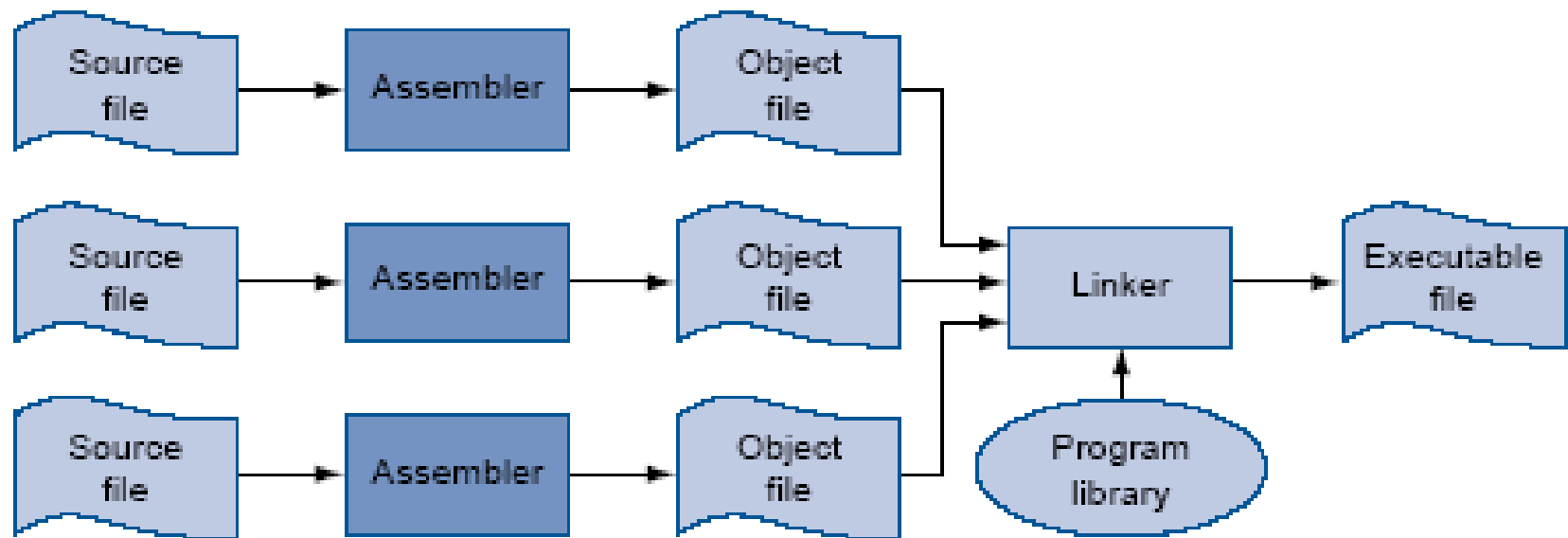
- MIPS has half-word instructions
  - lh \$t0,0(\$sp)
  - sh \$t0,0(\$sp)

# The Process

- Source Code => Assembly Code
- Ultimately, when a computer executes a program, it executes a series of machine language instructions.
  - In MIPS, that's one word (32 bits) per instruction
  - While not always true, in many cases, a compiler first translates a high level language program into assembly language



**FIGURE A.1.6** Assembly language either is written by a programmer or is the output of a compiler.



**FIGURE A.1.1** The process that produces an executable file. An assembler translates a file of assembly language into an object file, which is linked with other files and libraries into an executable file.

# Why Assembly Language?

- Embedded systems
- Time (execution) critical
- Space minimization

# Assembling a program

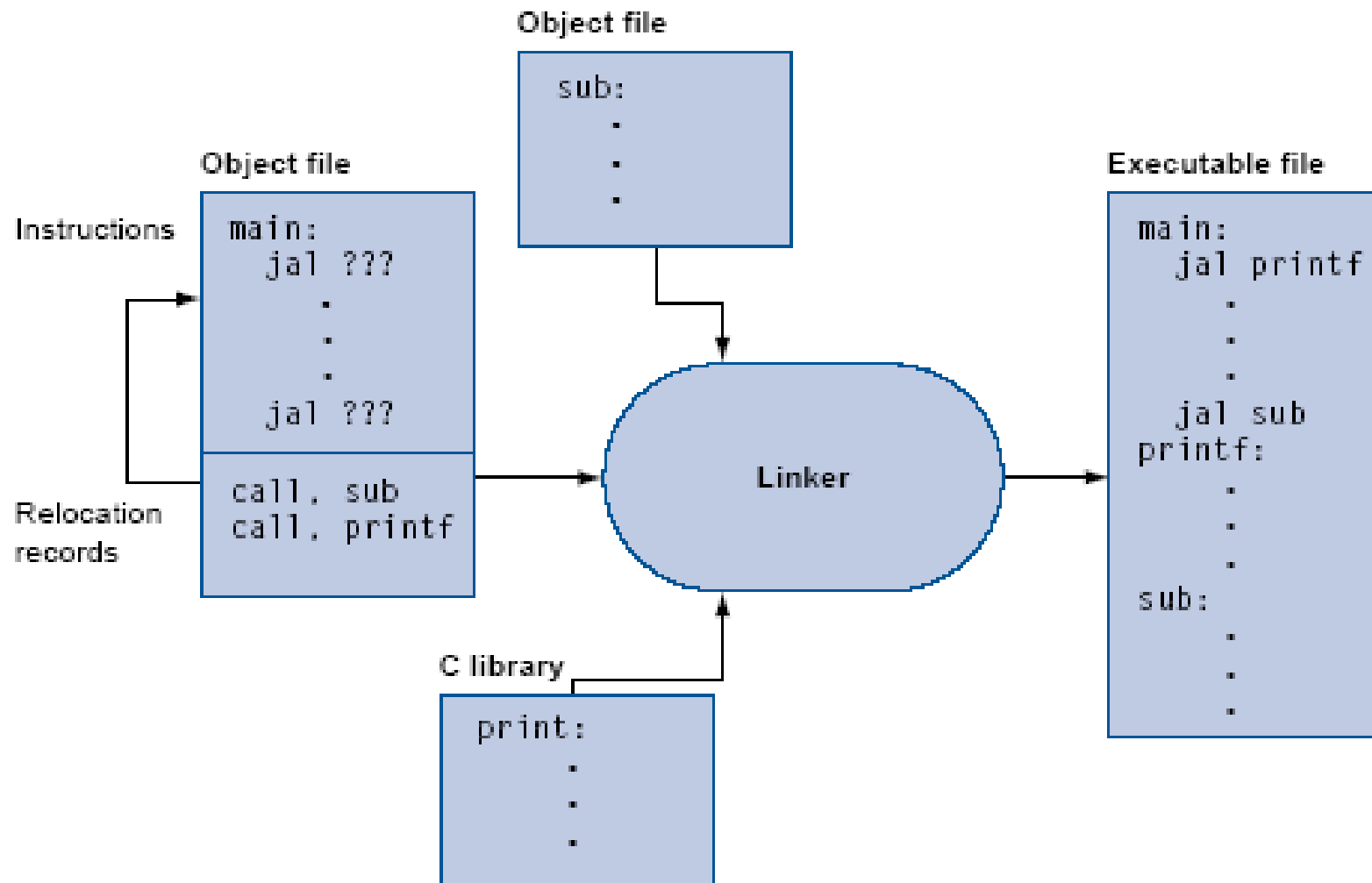
- Translate into machine language
- References
  - Forward references mean that the assembly process (on MIPS) is a two-pass process
  - Unresolved references are placed in a table for the linker to resolve
- The assembler actually produces a file called an object file – it will likely have unresolved references, i.e. references to programs in some other module

# Linker

- The linker “assembles” the various modules into executable code
  - Must resolve references to libraries and modules referenced from other modules
- Dynamically linked
  - Linking does not occur until program is loaded and only referenced modules are linked (loaded)
  - Another procedure only does link when code actually executes a call to the module

# Loader

- Actually loads an executable module into memory.
  - May be contiguous or paged
  - Each program has an offset register so that program references are  $0 + \text{offset amount}$



**FIGURE A.3.1** The linker searches a collection of object files and program libraries to find nonlocal routines used in a program, combines them into a single executable file, and resolves references between routines in different files.

# Compiler Optimization

- Procedure Inlining
- Loop unrolling
- Register optimization
  - Includes sub-expression optimization, i.e. substitute for a sub-expression the contents of a register holding that value
- Strength reduction – replace a complex instruction with a simpler one, e.g. shift left to replace multiply

# Compiler Optimization

- Constant folding, e.g.  $1+1 \Rightarrow$  replace with 2
- Dead store and code elimination
  - Eliminate stores of values that are not used again
  - Code that has no effect or cannot be reached is eliminated