# CS 110
# Computer Architecture
# *Running a Program - CALL (Compiling, Assembling, Linking, and Loading)*

Instructor:
**Sören Schwertfeger**

**http://shtech.org/courses/ca/**

**School of Information Science and Technology SIST**

**ShanghaiTech University**

**Slides based on UC Berkley's CS61C**

# Branch Calculation

- If we don't take the branch:
  - `PC = PC + 4` = next instruction
- If we do take the branch:
  - `PC = (PC+4) + (immediate*4)`

- **Observations:**
  - `immediate` is number of instructions to jump (remember, specifies words) either forward (+) or backwards (–)
  - Branch from `PC+4` for hardware reasons; will be clear why later in the course

# Branch Example (1/2)

- MIPS Code:

```
Loop: beq   $9,$0,End
      addu  $8,$8,$10
      addiu $9,$9,-1
      j     Loop
End:
```

Start counting from instruction AFTER the branch

1
2
3

- I-Format fields:

| | |
|---|---|
| `opcode` = 4 | (look up on Green Sheet) |
| `rs` = 9 | (first operand) |
| `rt` = 0 | (second operand) |
| `immediate` = 3 | |

# Branch Example (2/2)

- MIPS Code:

```
Loop: beq   $9,$0,End
      addu  $8,$8,$10
      addiu $9,$9,-1
      j     Loop
End:
```

Field representation (decimal):

31 … 0

| | | | |
|---|---|---|---|
| 4 | 9 | 0 | 3 |

Field representation (binary):

31 … 0

| | | | |
|---|---|---|---|
| 000100 | 01001 | 00000 | 0000000000000011 |

# J-Format Instructions (1/4)

- For branches, we assumed that we won't want to branch too far, so we can specify a *change* in the PC
- For general jumps (`j` and `jal`), we may jump to *anywhere* in memory
  - Ideally, we would specify a 32-bit memory address to jump to
  - Unfortunately, we can't fit both a 6-bit `opcode` and a 32-bit address into a single 32-bit word

# J-Format Instructions (2/4)

- Define two "fields" of these bit widths:

| 31 | 0 |
|---|---|
| 6 | 26 |

- As usual, each field has a name:

| 31 | 0 |
|---|---|
| `opcode` | `target address` |

- **Key Concepts:**
  - Keep `opcode` field identical to R-Format and I-Format for consistency
  - Collapse all other fields to make room for large target address

# J-Format Instructions (3/4)

- We can specify $2^{26}$ addresses
  - Still going to word-aligned instructions, so add `00` as last two bits (multiply by 4)
  - This brings us to 28 bits of a 32-bit address
- Take the 4 highest order bits from the PC
  - Cannot reach *everywhere*, but adequate almost all of the time, since programs aren't that long
  - Only problematic if code straddles a 256MB boundary
- If necessary, use 2 jumps or `jr` (R-Format) instead

# J-Format Instructions (4/4)

- Jump instruction:
  - New PC = { (PC+4)[31..28], target address, 00 }
- Notes:
  - { , , } means concatenation
    { 4 bits , 26 bits , 2 bits } = 32 bit address
    - Book uses | | instead
  - Array indexing: [31..28] means highest 4 bits
  - For hardware reasons, use PC+4 instead of PC

# Assembler Pseudo-Instructions

- Certain C statements are implemented unintuitively in MIPS
  - e.g. assignment (`a=b`) via add $zero
- MIPS has a set of "pseudo-instructions" to make programming easier
  - More intuitive to read, but get translated into actual instructions later
- Example:

```
move dst,src
```

translated into

```
addi dst,src,0
```

# Assembler Pseudo-Instructions

- List of pseudo-instructions: http://en.wikipedia.org/wiki/MIPS_architecture#Pseudo_instructions
  - List also includes instruction translation
- Load Address (`la`)
  - `la dst,label`
  - Loads address of specified label into `dst`
- Load Immediate (`li`)
  - `li dst,imm`
  - Loads 32-bit immediate into `dst`
- MARS has additional pseudo-instructions
  - See Help (F1) for full list

# Assembler Register

- Problem:
  - When breaking up a pseudo-instruction, the assembler may need to use an extra register
  - If it uses a regular register, it'll overwrite whatever the program has put into it
- Solution:
  - Reserve a register ($1 or $at for "assembler temporary") that assembler will use to break up pseudo-instructions
  - Since the assembler may use this at any time, it's not safe to code with it

# Multiply and Divide

- Example pseudo-instruction:

  **`mul $rd,$rs,$rt`**

  – Consists of mult which stores the output in special hi and lo registers, and a move from these registers to $rd

  ```
  mult $rs,$rt
  mflo $rd
  ```

- **`mult`** and **`div`** have nothing important in the **`rd`** field since the destination registers are **`hi`** and **`lo`**
- **`mfhi`** and **`mflo`** have nothing important in the **`rs`** and **`rt`** fields since the source is determined by the instruction (see COD)

# Integer Multiplication (1/3)

- Paper and pencil example (unsigned):

```
Multiplicand     1000  8
 Multiplier    x 1001  9
                 ----
                  1000
                +0000
               +0000
              +1000
              --------
              01001000  72
```

- m bits x n bits = m + n bit product

# Integer Multiplication (2/3)

- In MIPS, we multiply registers, so:
  - 32-bit value x 32-bit value = 64-bit value
- Syntax of Multiplication (signed):
  - `mult` register1, register2
  - Multiplies 32-bit values in those registers & puts 64-bit product in special result regs:
    - puts product upper half in hi, lower half in lo
  - hi and lo are 2 registers separate from the 32 general purpose registers
  - Use `mfhi` register & `mflo` register to move from hi, lo to another register

# Integer Multiplication (3/3)

- Example:
  - in C: `a = b * c;`
  - in MIPS:
    - let b be `$s2`; let c be `$s3`; and let a be `$s0` and `$s1` (since it may be up to 64 bits)

```
mult $s2,$s3  # b*c
mfhi $s0      # upper half of
              # product into $s0
mflo $s1      # lower half of
              # product into $s1
```

- Note: Often, we only care about the lower half of the product.
- Pseudo-inst. `mul` expands to `mult/mflo`.

# Integer Division (1/2)

- Paper and pencil example (unsigned):
  - 74 / 8 = 9  Rest 2

```
           1001       Quotient Divisor
      1000|1001010 Dividend
          -1000
              10
              101
              1010
             -1000
                10 Remainder
              (or Modulo result)
```

- Dividend = Quotient x Divisor + Remainder

# Integer Division (2/2)

- Syntax of Division (signed):
  - `div` register1, register2
  - Divides 32-bit register 1 by 32-bit register 2:
  - puts remainder of division in `hi`, quotient in `lo`
- Implements C division (`/`) and modulo (`%`)
- Example in C: `a = c / d;` `b = c % d;`
- in MIPS: `a↔$s0;b↔$s1;c↔$s2;d↔$s3`

```
div  $s2,$s3      # lo=c/d, hi=c%d
mflo $s0          # get quotient
mfhi $s1          # get remainder
```

# MAL vs. TAL

- True Assembly Language (TAL)
  - The instructions a computer understands and executes

- MIPS Assembly Language (MAL)
  - Instructions the assembly programmer can use (includes pseudo-instructions)
  - Each MAL instruction becomes 1 or more TAL instruction

# Summary

- I-Format: instructions with immediates, `lw`/`sw` (offset is immediate), and `beq`/`bne`
  - But not the shift instructions
  - Branches use PC-relative addressing

**I:**

| opcode | rs | rt | immediate |
|---|---|---|---|

- J-Format: `j` and `jal` (but not `jr`)
  - Jumps use absolute addressing

**J:**

| opcode | target address |
|---|---|

- R-Format: all other instructions

**R:**

| opcode | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|

# Levels of Representation/Interpretation

High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

*Compiler*

Assembly Language Program (e.g., MIPS)

```
lw   $t0, 0($2)
lw   $t1, 4($2)
sw   $t1, 0($2)
sw   $t0, 4($2)
```

*Assembler*

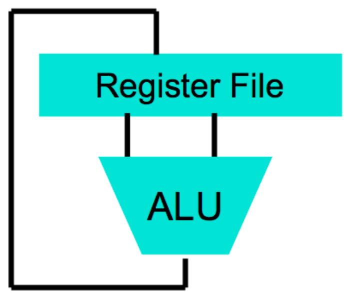
Machine Language Program (MIPS)

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

Anything can be represented as a *number*, i.e., data or instructions

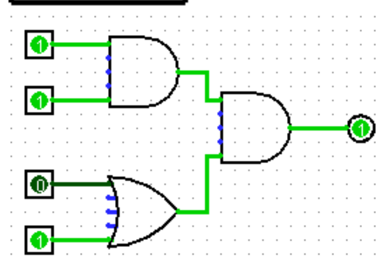
*Machine Interpretation*

Hardware Architecture Description (e.g., block diagrams)




*Architecture Implementation*

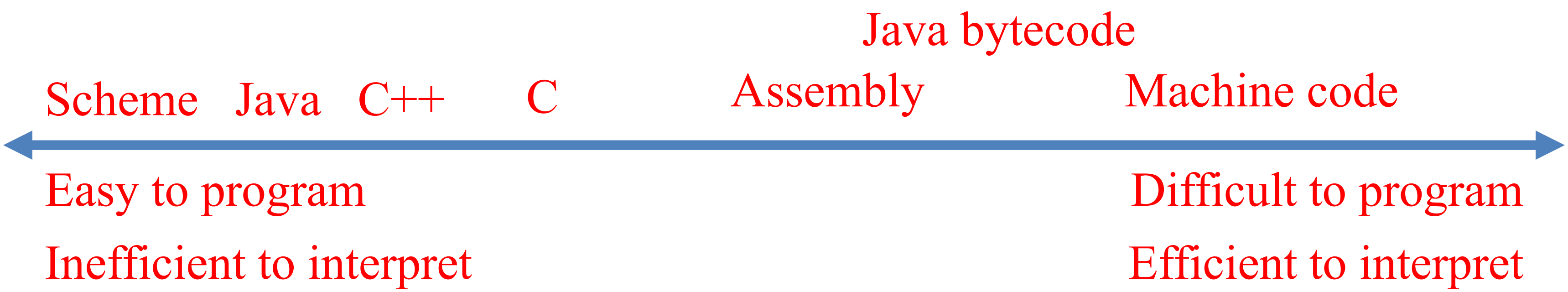
Logic Circuit Description (Circuit Schematic Diagrams)



+ How to take a program and run it

# Language Execution Continuum

- An Interpreter is a program that executes other programs.

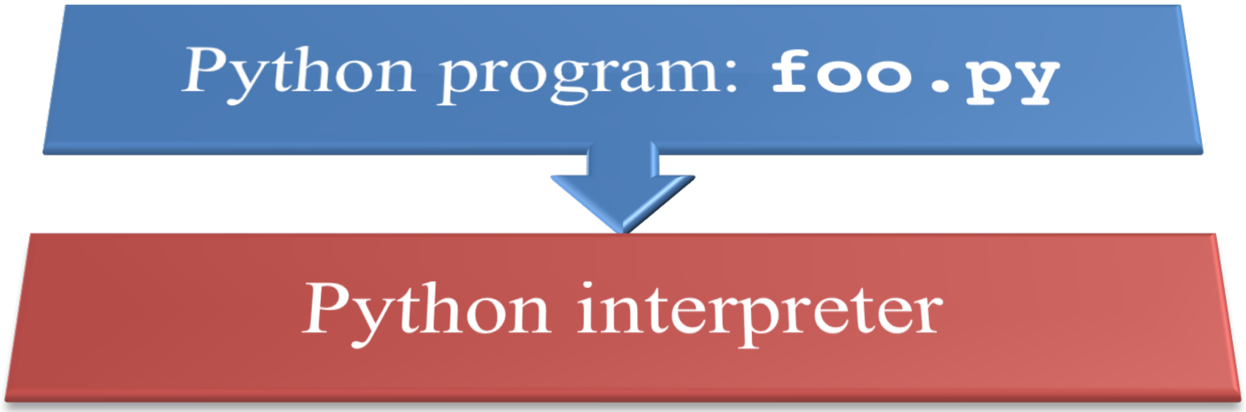



- Language translation gives us another option
- In general, we interpret a high-level language when efficiency is not critical and translate to a lower-level language to increase performance

# Interpretation vs Translation

- How do we run a program written in a source language?
  - Interpreter: Directly executes a program in the source language
  - Translator: Converts a program from the source language to an equivalent program in another language
- For example, consider a Python program **`foo.py`**

# Interpretation




- Python interpreter is just a program that reads a python program and performs the functions of that python program.

# Interpretation

- Any good reason to interpret machine language in software?
- MARS– useful for learning / debugging
- Apple Macintosh conversion
  - Switched from Motorola 680x0 instruction architecture to PowerPC.
    - Similar issue with switch to x86
  - Could require all programs to be re-translated from high level language
  - Instead, let executables contain old and/or new machine code, interpret old code in software if necessary (emulation)

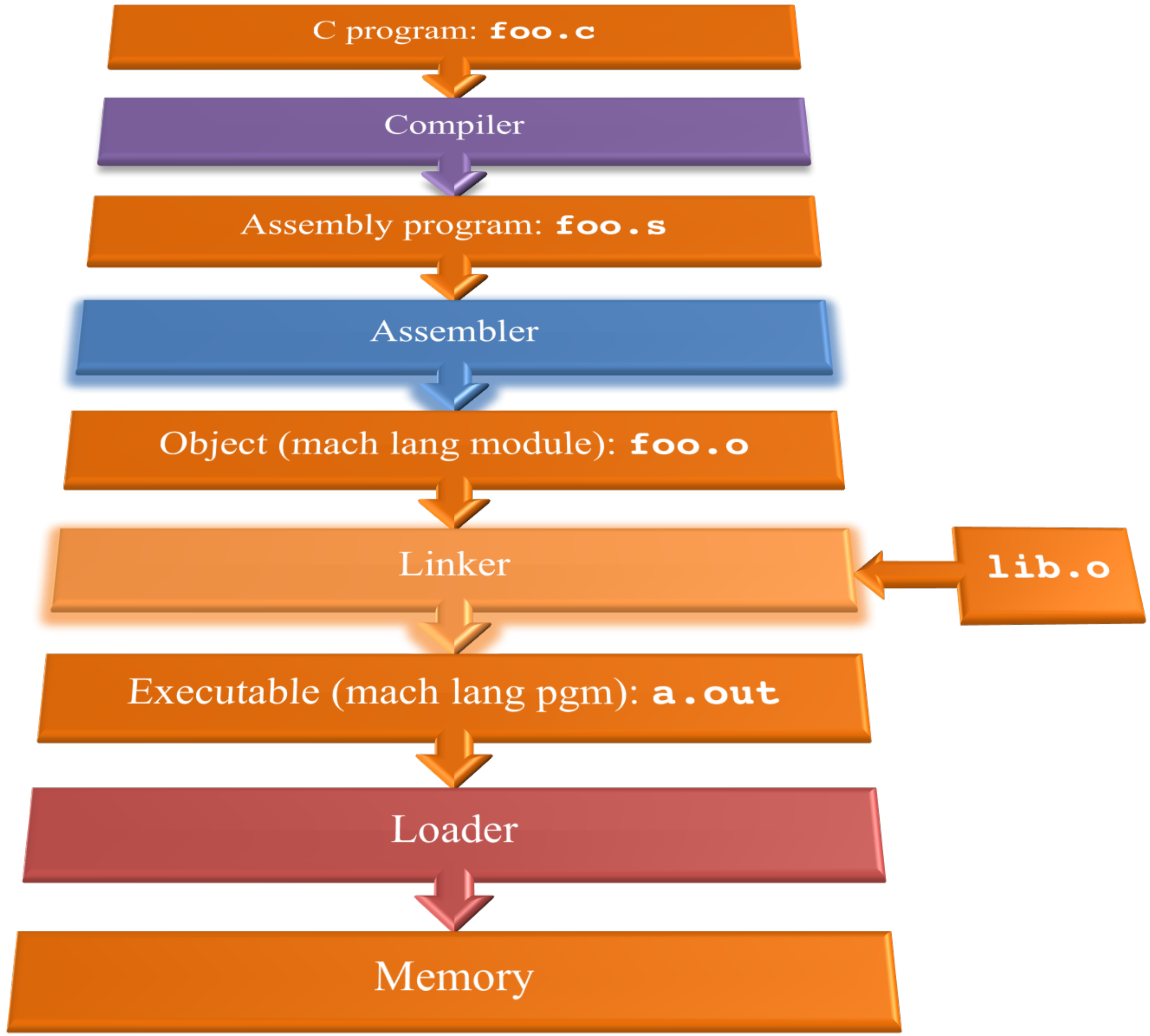
# Interpretation vs. Translation? (1/2)

- Generally easier to write interpreter
- Interpreter closer to high-level, so can give better error messages (e.g., MARS)
  - Translator reaction: add extra information to help debugging (line numbers, names)
- Interpreter slower (10x?), code smaller (2x?)
- Interpreter provides instruction set independence: run on any machine

# Interpretation vs. Translation? (2/2)

- Translated/compiled code almost always more efficient and therefore higher performance:
  - Important for many applications, particularly operating systems.
- Translation/compilation helps "hide" the program "source" from the users:
  - One model for creating value in the marketplace (eg. Microsoft keeps all their source code secret)
  - Alternative model, "open source", creates value by publishing the source code and fostering a community of developers.

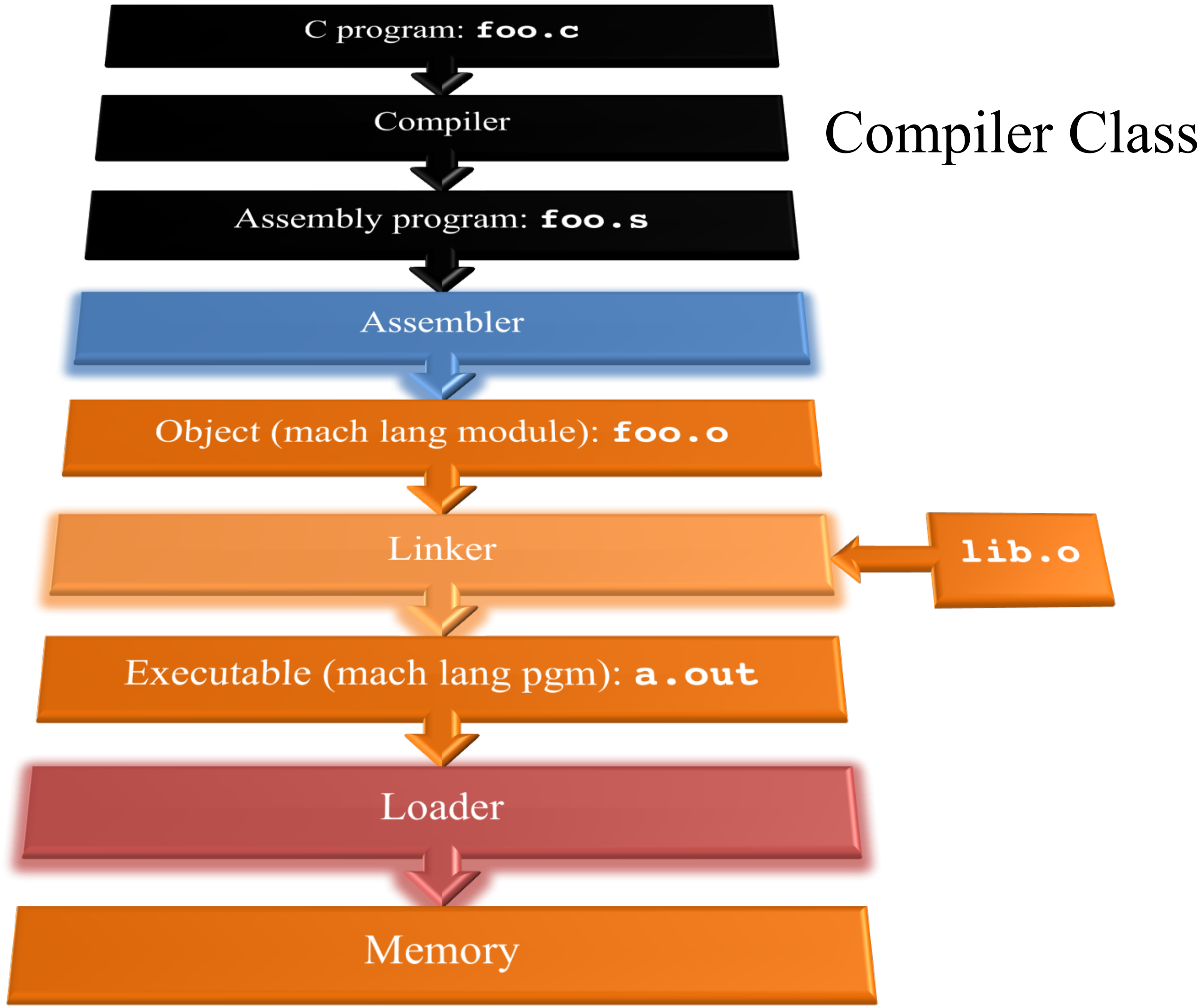
# Steps in compiling a C program



# Compiler

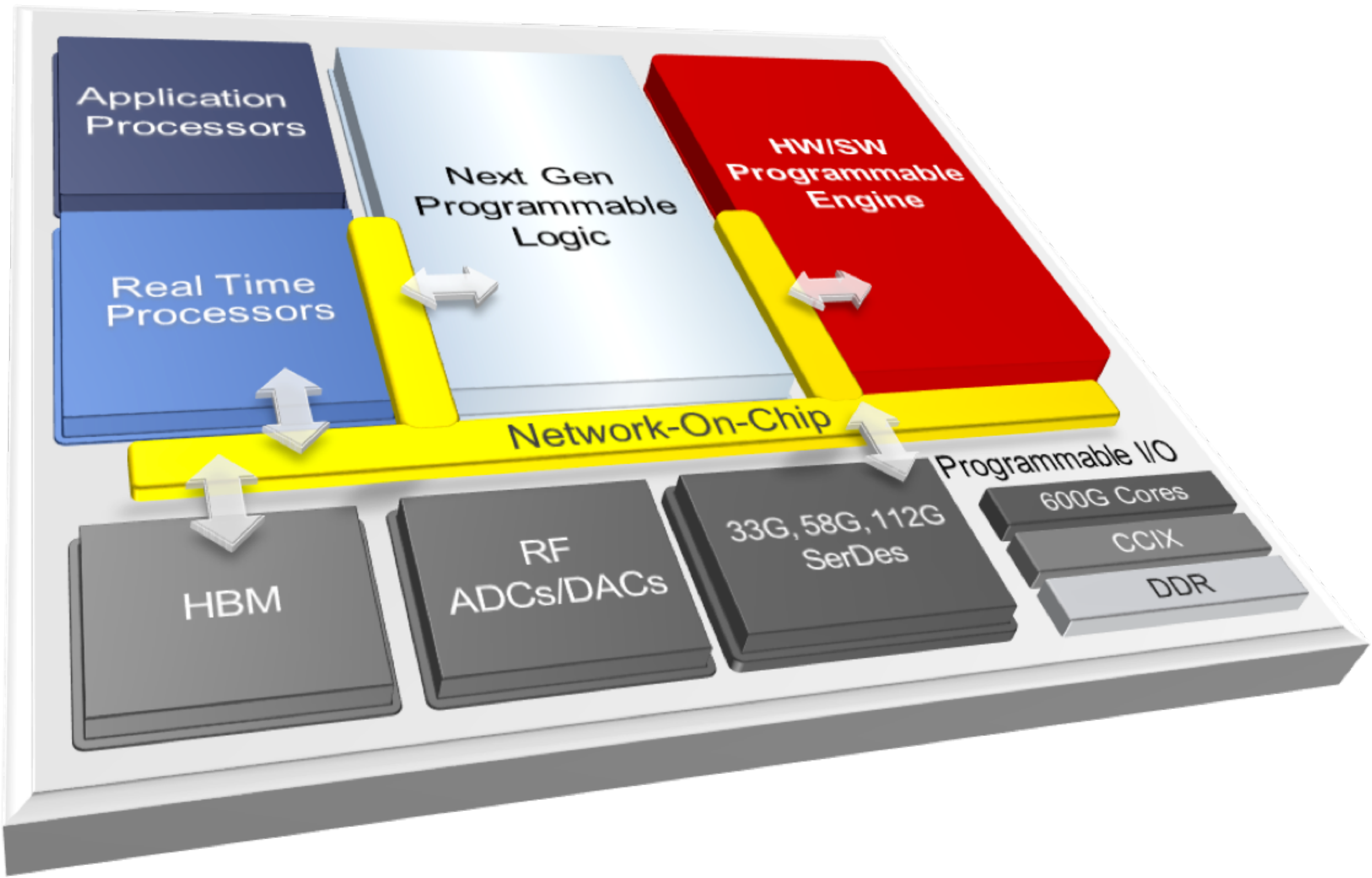
- Input: High-Level Language Code (e.g., **`foo.c`**)
- Output: Assembly Language Code (e.g., **`foo.s`** for MIPS)
- Note: Output *may* contain pseudo-instructions
- Pseudo-instructions: instructions that assembler understands but not in machine For example:
  - **`move $s1,$s2`** ⇒ **`add $s1,$s2,$zero`**

# Where Are We Now?



# In the news: Xilinx ACAP

- Adaptive Compute Acceleration Platform
  - Advanced form of FPGA
  - Machine Learning, Genomic, Video-coding, Data-analysis, Fintec
  - 50 billion transistors
  - 7nm (first chip in 7nm!)
  - 10-100x speed vs. CPUs
  - Faster than GPU
  - Much more energy efficient
  - For datacenters / supercomputing


Application Processors
Next Gen Programmable Logic
HW/SW Programmable Engine
Real Time Processors
Network-On-Chip
Programmable I/O
600G Cores
CCIX
DDR
HBM
RF ADCs/DACs
33G, 58G, 112G SerDes

# Assembler

- Input: Assembly Language Code (MAL) (e.g., **`foo.s`** for MIPS)
- Output: Object Code, information tables (TAL) (e.g., **`foo.o`** for MIPS)
- Reads and Uses Directives
- Replace Pseudo-instructions
- Produce Machine Language
- Creates Object File

# Assembler Directives (p. A-13.. A-17)

- Give directions to assembler, but do not produce machine instructions

  **`.text`:** Subsequent items put in user text segment (machine code)

  **`.data`:** Subsequent items put in user data segment (binary rep of data in source file)

  **`.globl sym`:** declares `sym` global and can be referenced from other files

  **`.asciiz str`:** Store the string **`str`** in memory and null-terminate it

  **`.word w1…wn`:** Store the *n* 32-bit quantities in successive memory words

# Pseudo-instruction Replacement

- Assembler treats convenient variations of machine language instructions as if real instructions

| Pseudo: | Real: |
|---|---|
| `subu $sp,$sp,32` | `addiu $sp,$sp,-32` |
| `sd $a0, 32($sp)` | `sw $a0, 32($sp)`<br>`sw $a1, 36($sp)` |
| `mul $t7,$t6,$t5` | `mult $t6,$t5`<br>`mflo $t7` |
| `addu $t0,$t6,1` | `addiu $t0,$t6,1` |
| `ble $t0,100,loop` | `slti $at,$t0,101`<br>`bne $at,$0,loop` |
| `la $a0, str` | `lui $at,left(str)`<br>`ori $a0,$at,right(str)` |

# Question

Which of the following is a correct TAL instruction sequence for la $v0, FOO?*

%hi(label), tells assembler to fill upper 16 bits of label's addr

%lo(label), tells assembler to fill lower 16 bits of label's addr

A: ori $v0, %hi(FOO)
    addiu $v0, %lo(FOO)

B: ori $v0, %lo(FOO)
    lui $v0, %hi(FOO)

C: lui $v0, %lo(FOO)
    ori $v0, %hi(FOO)

D: lui $v0, %hi(FOO)
    ori $v0, %lo(FOO)

E: la $v0, FOO is already a TAL instruction

*Assume the address of FOO is 0xABCD0124

# Producing Machine Language (1/3)

- Simple Case
  - Arithmetic, Logical, Shifts, and so on
  - All necessary info is within the instruction already
- What about Branches?
  - PC-Relative
  - So once pseudo-instructions are replaced by real ones, we know by how many instructions to branch
- So these can be handled

# Producing Machine Language (2/3)

- "Forward Reference" problem
  - Branch instructions can refer to labels that are "forward" in the program:

```
    or   $v0, $0,  $0
L1: slt  $t0, $0,  $a1
    beq  $t0, $0,  L2
    addi $a1, $a1, -1
    j    L1
L2: add  $t1, $a0, $a1
```

  - Solved by taking 2 passes over the program
    - First pass remembers position of labels
    - Second pass uses label positions to generate code

# Producing Machine Language (3/3)

- What about jumps (**`j`** and **`jal`**)?
  - Jumps require absolute address
  - So, forward or not, still can't generate machine instruction without knowing the position of instructions in memory
- What about references to static data?
  - **`la`** gets broken up into **`lui`** and **`ori`**
  - These will require the full 32-bit address of the data
- These can't be determined yet, so we create two tables…

# Symbol Table

- List of "items" in this file that may be used by other files
- What are they?
  - Labels: function calling
  - Data: anything in the `.data` section; variables which may be accessed across files

# Relocation Table

- List of "items" this file needs the address of later
- What are they?
  - Any label jumped to: **`j`** or **`jal`**
    - internal
    - external (including lib files)
  - Any piece of data in static section
    - such as the **`la`** instruction

# Object File Format

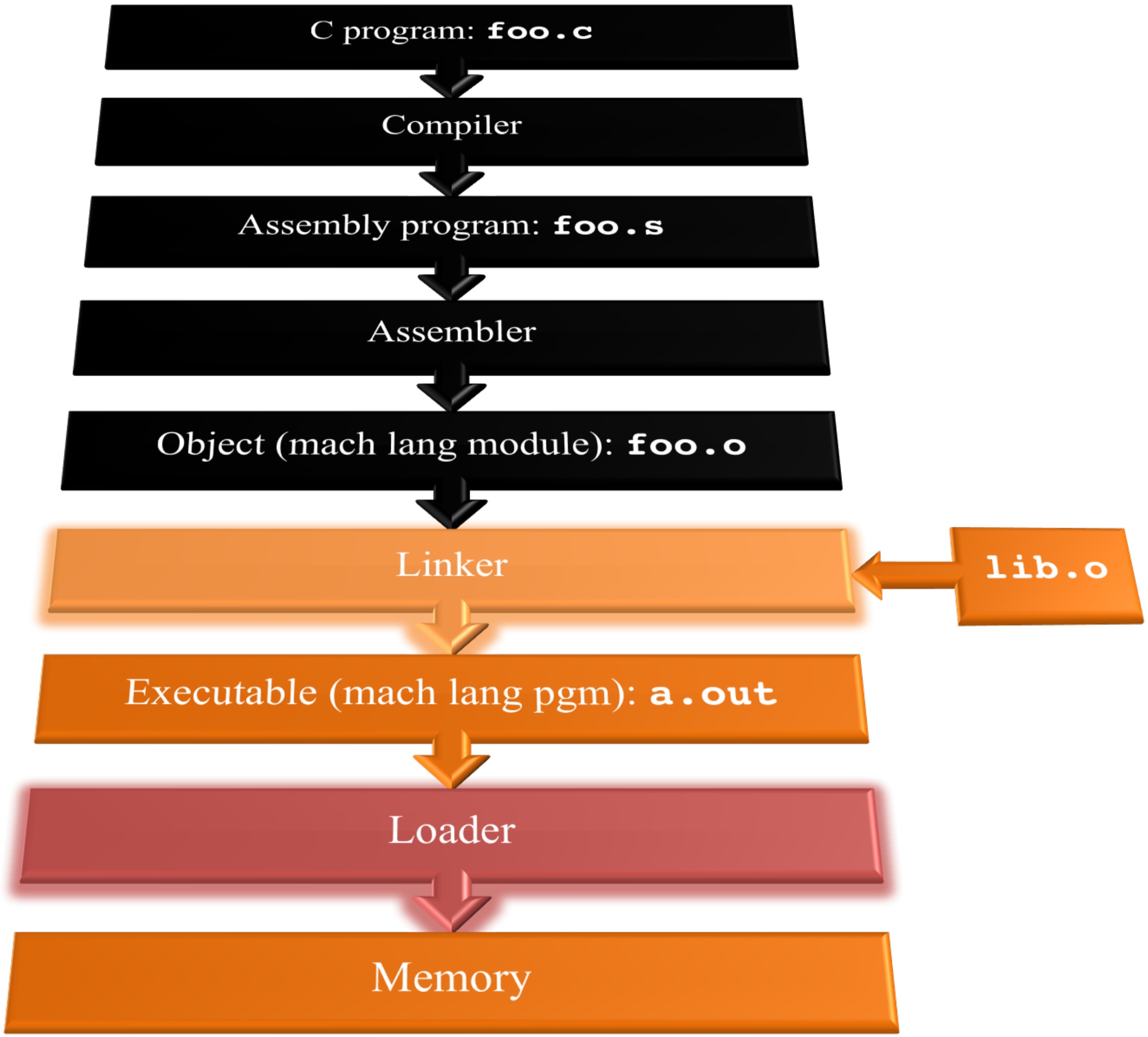
- object file header: size and position of the other pieces of the object file
- text segment: the machine code
- data segment: binary representation of the static data in the source file
- relocation information: identifies lines of code that need to be fixed up later
- symbol table: list of this file's labels and static data that can be referenced
- debugging information
- A standard format is ELF (except MS)

http://www.skyfree.org/linux/references/ELF_Format.pdf

# Admin

- Project 1.1 will be published today (this time for real ;) )

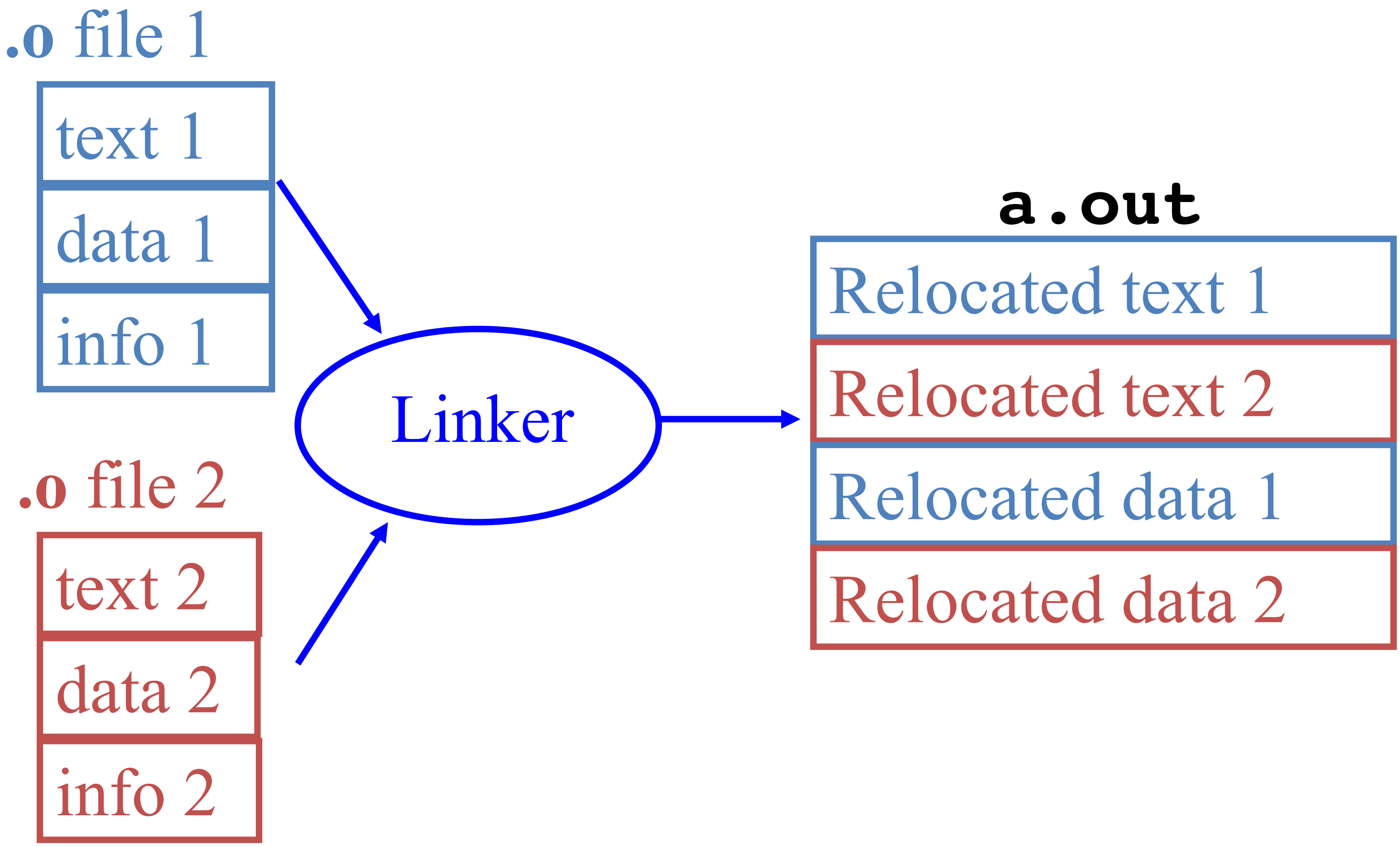
# Where Are We Now?



# Linker (1/3)

- Input: Object code files, information tables (e.g., `foo.o,libc.o` for MIPS)
- Output: Executable code (e.g., `a.out` for MIPS)
- Combines several object (`.o`) files into a single executable ("linking")
- Enable separate compilation of files
  - Changes to one file do not require recompilation of the whole program
    - Windows NT source was > 40 M lines of code!
  - Old name "Link Editor" from editing the "links" in jump and link instructions

# Linker (2/3)


.o file 1
text 1
data 1
info 1
.o file 2
text 2
data 2
info 2
Linker
a.out
Relocated text 1
Relocated text 2
Relocated data 1
Relocated data 2

# Linker (3/3)

- Step 1: Take text segment from each `.o` file and put them together
- Step 2: Take data segment from each `.o` file, put them together, and concatenate this onto end of text segments
- Step 3: Resolve references
  - Go through Relocation Table; handle each entry
  - That is, fill in all absolute addresses

# Four Types of Addresses

- PC-Relative Addressing (`beq`, `bne`)
  - never relocate
- Absolute Function Address (`j`, `jal`)
  - always relocate
- External Function Reference (usually `jal`)
  - always relocate
- Static Data Reference (often `lui` and `ori`)
  - always relocate

# Absolute Addresses in MIPS

- Which instructions need relocation editing?
  - J-format: jump, jump and link

| | |
|---|---|
| **j/jal** | **xxxxx** |

  - Loads and stores to variables in static area, relative to global pointer

| | | | |
|---|---|---|---|
| **lw/sw** | **$gp** | **$x** | **address** |

  - What about conditional branches?

| | | | |
|---|---|---|---|
| **beq/bne** | **$rs** | **$rt** | **address** |

  - PC-relative addressing preserved even if code moves

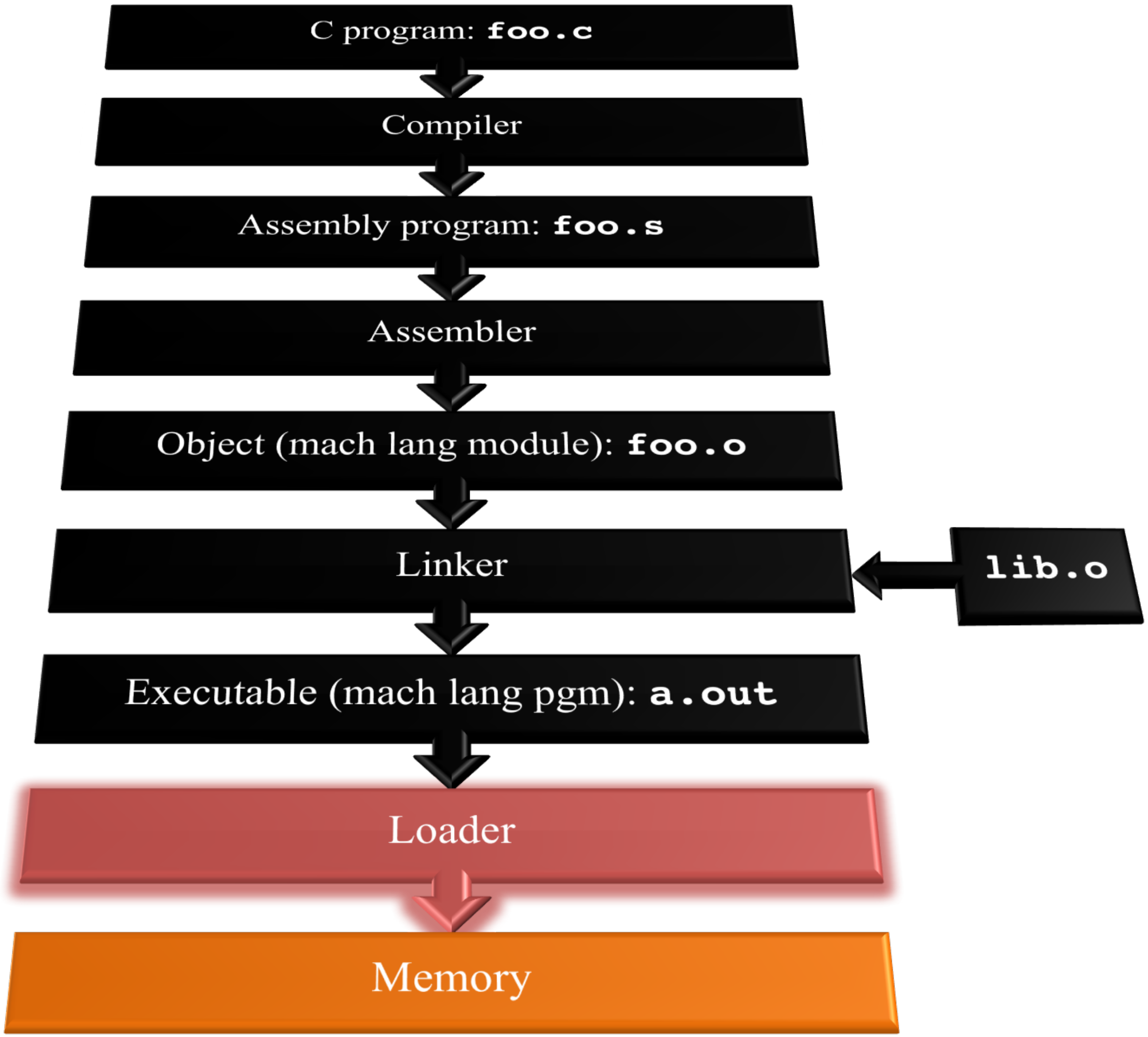
# Resolving References (1/2)

- Linker assumes first word of first text segment is at address **`0x04000000`**.
  - (More later when we study “virtual memory”)
- Linker knows:
  - length of each text and data segment
  - ordering of text and data segments
- Linker calculates:
  - absolute address of each label to be jumped to (internal or external) and each piece of data being referenced

# Resolving References (2/2)

- To resolve references:
  - search for reference (data or label) in all "user" symbol tables
  - if not found, search library files (for example, for `printf`)
  - once absolute address is determined, fill in the machine code appropriately
- Output of linker: executable file containing text and data (plus header)

# Where Are We Now?



# Loader Basics

- Input: Executable Code
  (e.g., **`a.out`** for MIPS)
- Output: (program is run)
- Executable files are stored on disk
- When one is run, loader's job is to load it into memory and start it running
- In reality, loader is the operating system (OS)
  – loading is one of the OS tasks

# Loader … what does it do?

- Reads executable file's header to determine size of text and data segments
- Creates new address space for program large enough to hold text and data segments, along with a stack segment
- Copies instructions and data from executable file into the new address space
- Copies arguments passed to the program onto the stack
- Initializes machine registers
  - Most registers cleared, but stack pointer assigned address of 1st free stack location
- Jumps to start-up routine that copies program's arguments from stack to registers & sets the PC
  - If main routine returns, start-up routine terminates program with the exit system call

# Question

At what point in process are all the machine code bits generated for the following assembly instructions:

1) `addu $6, $7, $8`

2) `jal fprintf`

A: 1) & 2) After compilation

B: 1) After compilation, 2) After assembly

C: 1) After assembly, 2) After linking

D: 1) After assembly, 2) After loading

E: 1) After compilation, 2) After linking

# Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

*C Program Source Code:* ***prog.c***

```
#include <stdio.h>
int main (int argc, char *argv[]) {
 int i, sum = 0;
 for (i = 0; i <= 100; i++)
    sum = sum + i * i;
 printf ("The sum of sq from 0 .. 100 is
  %d\n",    sum);
}
```

*"**printf**" lives in "**libc**"*

# Compilation: MAL

```
   .text
   .align  2
   .globl  main
main:
   subu $sp,$sp,32
   sw $ra, 20($sp)
   sd $a0, 32($sp)
   sw $0, 24($sp)
   sw $0, 28($sp)
loop:
   lw $t6, 28($sp)
   mul $t7, $t6,$t6
   lw $t8, 24($sp)
   addu $t9,$t8,$t7
   sw $t9, 24($sp)
```

```
   addu $t0, $t6, 1
   sw $t0, 28($sp)
   ble $t0,100, loop
   la $a0, str
   lw $a1, 24($sp)
   jal printf
   move $v0, $0
   lw $ra, 20($sp)
   addiu $sp,$sp,32
   jr $ra
   .data
   .align  0
str:
   .asciiz "The sum
   of sq from 0 ..
   100 is %d\n"
```

**Where are 7 pseudo-instructions?**

# Compilation: MAL

```
   .text
   .align   2
   .globl   main
main:
   subu $sp,$sp,32
   sw $ra, 20($sp)
   sd $a0, 32($sp)
   sw $0, 24($sp)
   sw $0, 28($sp)
loop:
   lw $t6, 28($sp)
   mul $t7, $t6,$t6
   lw $t8, 24($sp)
   addu $t9,$t8,$t7
   sw $t9, 24($sp)
   addu $t0, $t6, 1
   sw $t0, 28($sp)
   ble $t0,100, loop
   la $a0, str
   lw $a1, 24($sp)
   jal printf
   move $v0, $0
   lw $ra, 20($sp)
   addiu $sp,$sp,32
   jr $ra
   .data
   .align   0
str:
   .asciiz "The sum
   of sq from 0 ..
   100 is %d\n"
```

**7 pseudo-instructions underlined**

# Assembly step 1:

Remove pseudoinstructions, assign addresses

```
00 addiu  $29, $29,-32
04 sw     $31,  20($29)
08 sw     $4,   32($29)
0c sw     $5,   36($29)
10 sw     $0,   24($29)
14 sw     $0,   28($29)
18 lw     $14,  28($29)
1c multu  $14, $14
20 mflo   $15
24 lw     $24,  24($29)
28 addu   $25, $24,$15
2c sw     $25,  24($29)
```

```
30 addiu  $8, $14, 1
34 sw     $8,   28($29)
38 slti   $1, $8, 101
3c bne    $1, $0, loop
40 lui    $4,     l.str
44 ori    $4, $4,r.str
48 lw     $5,   24($29)
4c jal    printf
50 add    $2, $0, $0
54 lw     $31,  20($29)
58 addiu  $29, $29,32
5c jr $31
```

# Assembly step 2

**Create relocation table and symbol table**

- Symbol Table

| Label | address (in module) | type |
|---|---|---|
| `main:` | `0x00000000` | `global text` |
| `loop:` | `0x00000018` | `local text` |
| `str:` | `0x00000000` | `local data` |

- Relocation Information

| Address | Instr. type | Dependency |
|---|---|---|
| `0x00000040` | `lui` | `l.str` |
| `0x00000044` | `ori` | `r.str` |
| `0x0000004c` | `jal` | `printf` |

# Assembly step 3

**Resolve local PC-relative labels**

```
00 addiu $29,$29,-32
04 sw    $31,20($29)
08 sw    $4, 32($29)
0c sw    $5, 36($29)
10 sw    $0, 24($29)
14 sw    $0, 28($29)
18 lw    $14, 28($29)
1c multu $14, $14
20 mflo  $15
24 lw    $24, 24($29)
28 addu  $25,$24,$15
2c sw    $25, 24($29)
```

```
30 addiu $8,$14, 1
34 sw    $8,28($29)
38 slti  $1,$8, 101
3c bne   $1,$0, -10
40 lui   $4, l.str
44 ori   $4,$4,r.str
48 lw    $5,24($29)
4c jal   printf
50 add   $2, $0, $0
54 lw    $31,20($29)
58 addiu $29,$29,32
5c jr    $31
```

# Assembly step 4

- Generate object (`.o`) file:
  - Output binary representation for
    - text segment (instructions)
    - data segment (data)
    - symbol and relocation tables
  - Using dummy "placeholders" for unresolved absolute and external references

# Text segment in object file

| | |
|---|---|
| 0x000000 | 00100111101111011111111111100000 |
| 0x000004 | 10101111101111110000000000010100 |
| 0x000008 | 10101111101001000000000000100000 |
| 0x00000c | 10101111101001010000000000100100 |
| 0x000010 | 10101111101000000000000000011000 |
| 0x000014 | 10101111101000000000000000011100 |
| 0x000018 | 10001111101011100000000000011100 |
| 0x00001c | 10001111101110000000000000011000 |
| 0x000020 | 00000001110011100000000000011001 |
| 0x000024 | 00100101110010000000000000000001 |
| 0x000028 | 00101001000000010000000001100101 |
| 0x00002c | 10101111101010000000000000011100 |
| 0x000030 | 00000000000000000111100000010010 |
| 0x000034 | 00000011000011111100100000100001 |
| 0x000038 | 00010100001000001111111111110111 |
| 0x00003c | 10101111101110010000000000011000 |
| 0x000040 | 00111100000001000000000000000000 |
| 0x000044 | 10001111101001010000000000000000 |
| 0x000048 | 00001100000100000000000011101100 |
| 0x00004c | 00100100000000000000000000000000 |
| 0x000050 | 10001111101111110000000000010100 |
| 0x000054 | 00100111101111010000000000100000 |
| 0x000058 | 00000011111000000000000000001000 |
| 0x00005c | 00000000000000000001000000100001 |

# Link step 1: combine `prog.o`, `libc.o`

- Merge text/data segments
- Create absolute memory addresses
- Modify & merge symbol and relocation tables
- Symbol Table
  - Label / Address

| Label | Address |
|---|---|
| `main:` | `0x00000000` |
| `loop:` | `0x00000018` |
| `str:` | `0x10000430` |
| `printf:` | `0x000003b0` |

…

- Relocation Information

| Address | Instr. Type | Dependency |
|---|---|---|
| `0x00000040` | `lui` | `l.str` |
| `0x00000044` | `ori` | `r.str` |
| `0x0000004c` | `jal` | `printf` |

…

# Link step 2:

- Edit Addresses in relocation table
  - (shown in TAL for clarity, but done in binary )

```
00 addiu $29,$29,-32
04 sw$31,20($29)
08 sw$4, 32($29)
0c sw$5, 36($29)
10 sw     $0, 24($29)
14 sw     $0, 28($29)
18 lw    $14, 28($29)
1c multu $14, $14
20 mflo $15
24 lw    $24, 24($29)
28 addu $25,$24,$15
2c sw    $25, 24($29)
```

```
30 addiu $8,$14, 1
34 sw$8,28($29)
38 slti $1,$8, 101
3c bne  $1,$0, -10
40 lui  $4, 4096
44 ori  $4,$4,1072
48 lw$5,24($29)
4c jal  944
50 add  $2, $0, $0
54 lw       $31,20($29)
58 addiu    $29,$29,32
5c jr$31
```

# Link step 3:

- Output executable of merged modules
  - Single text (instruction) segment
  - Single data segment
  - Header detailing size of each segment

- NOTE:
  - The preceding example was a much simplified version of how ELF and other standard formats work, meant only to demonstrate the basic principles.