CSC 416/565: DESIGN AND CONSTRUCTION OF COMPILERS

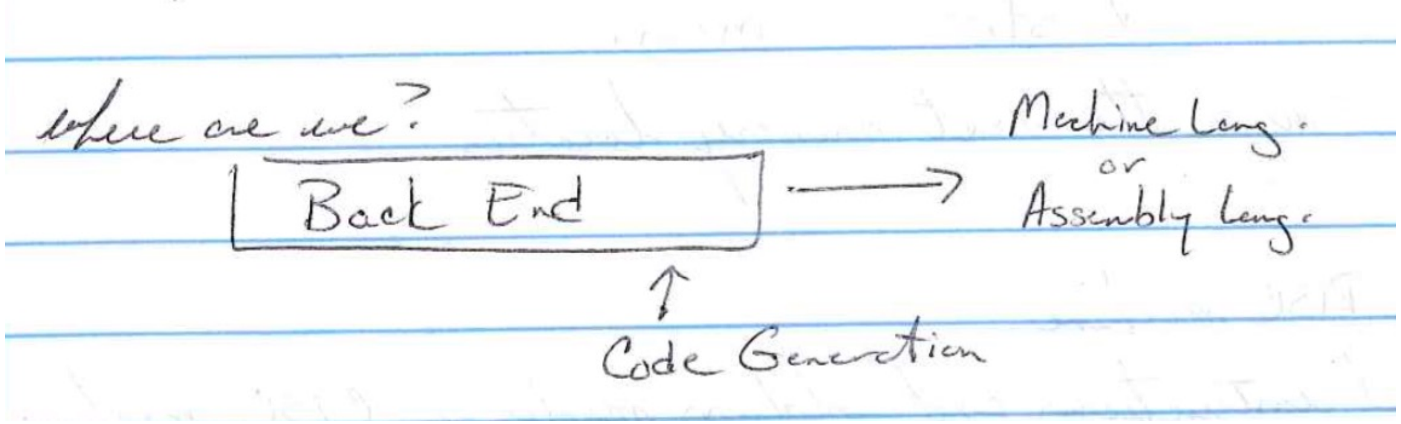
West Chester University
Dr. Richard Burns
Fall 2023

Code Generation I

References:

1. [Thain]
2. [Hennessy / Larus], *Computer Architecture, A Quantitative Approach* Appendix A
3. online MIPS references
4. MIPS simulator: MARS

Where Are We?



Will "translate" our AST/IR into <u>assembly language</u>.

- Very similar to machine language (binary 1s and 0s).
- Assembly is *symbolic*:
    1. symbols: names for commonly occurring bit patterns
    2. opcodes, register specifiers
    3. labels are allowed

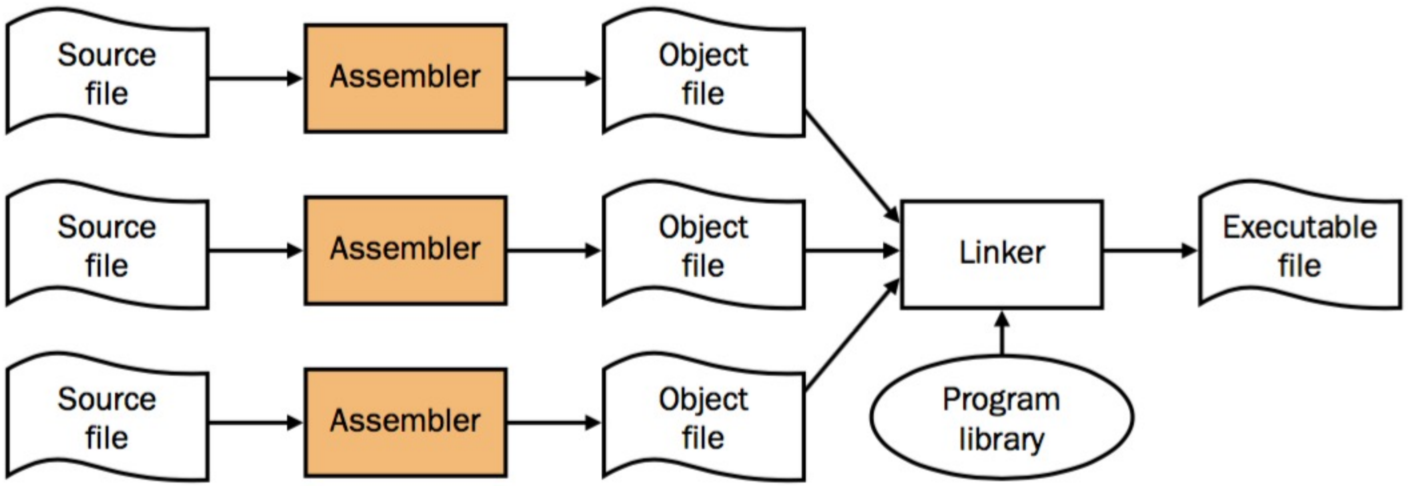


FIGURE A.1 The process that produces an executable file. An assembler translates a file of assembly language into an object file, which is linked with other files and libraries into an executable file.

- Assembler: translates assembly into binary
  - *input:* source file (in assembly)
  - *output:* object file (may reference subroutines and data in other object and library files)
- Linker: creates executable file by combining object and library files.

Assembler Directives

- Assembler Directives begin with a period
  - instructs the assembler how to translate the next part of the program
- *Examples:*

```
.data
.globl
.asciiz
```

`.text` assembler directive
indicates that succeeding lines contain instructions

`.data` assembler directive
indicates that succeeding lines contain data

`.asciiz` assembler directive
stores a null terminated string in memory; usually used w/ a label

`.globl` label
declares the label as a global symbol that should be visible to code stored in other files

```
        .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)
        addu    $sp, $sp, 32
        jr      $ra

        .data
        .align  0
str:
        .asciiz "The sum from 0 .. 100 is %d\n"
```

Labels

Labels are followed by a colon

- provide a symbolic name to the next memory location
- *Examples:*

```
str:
main:
```

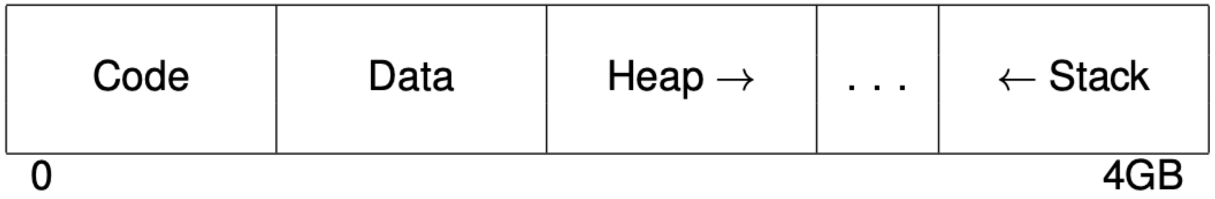
```
        .text
        .align  2
        .globl  main
main:
        subu    $sp, $sp, 32
        sw      $ra, 20($sp)
        sd      $a0, 32($sp)
        sw      $0,  24($sp)
        sw      $0,  28($sp)
loop:
        lw      $t6, 28($sp)
        mul     $t7, $t6, $t6
        lw      $t8, 24($sp)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

        .data
        .align  0
str:
        .asciiz "The sum from 0 .. 100 is %d\n"
```

Local vs. Global (External Labels)

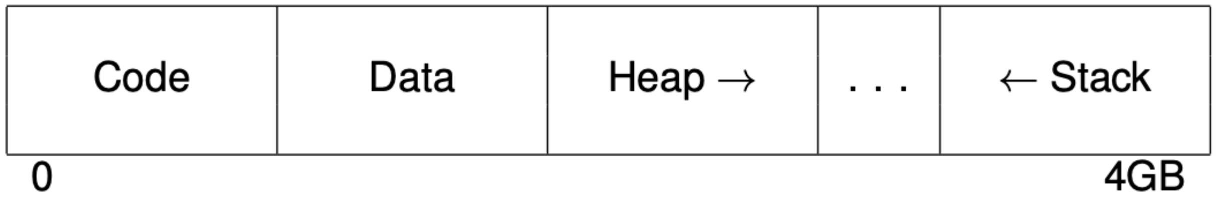
- Labels are local by default.
- Labels must be explicitly declared global.
- A label is external if the labeled object can be referenced from other files.

Loading a Program Into Memory and Executing It

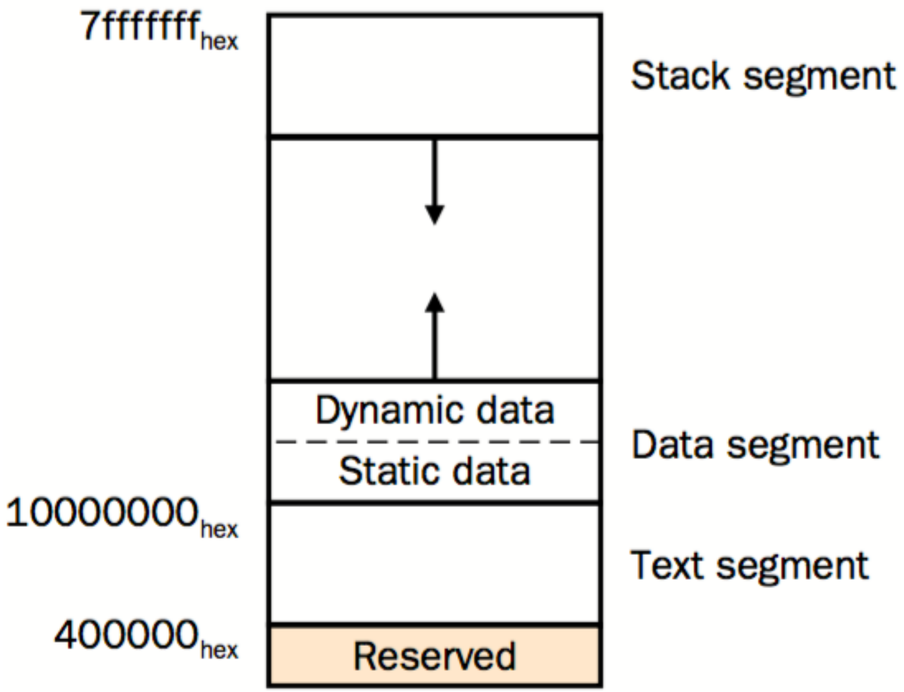


<u>Tasks:</u>

1. Creates new address space for the program; large enough to hold text and data segments.
2. Copies instructions and data into new address spaces; pushes program arguments onto stack.
3. Initializes (clears) machine registers.
   - $sp becomes address of first free stack location
4. Jump to the start-up routine.
   - copy program args from stack into registers
   - call program's main module
   - when main returns, return to terminal w/ exit system call

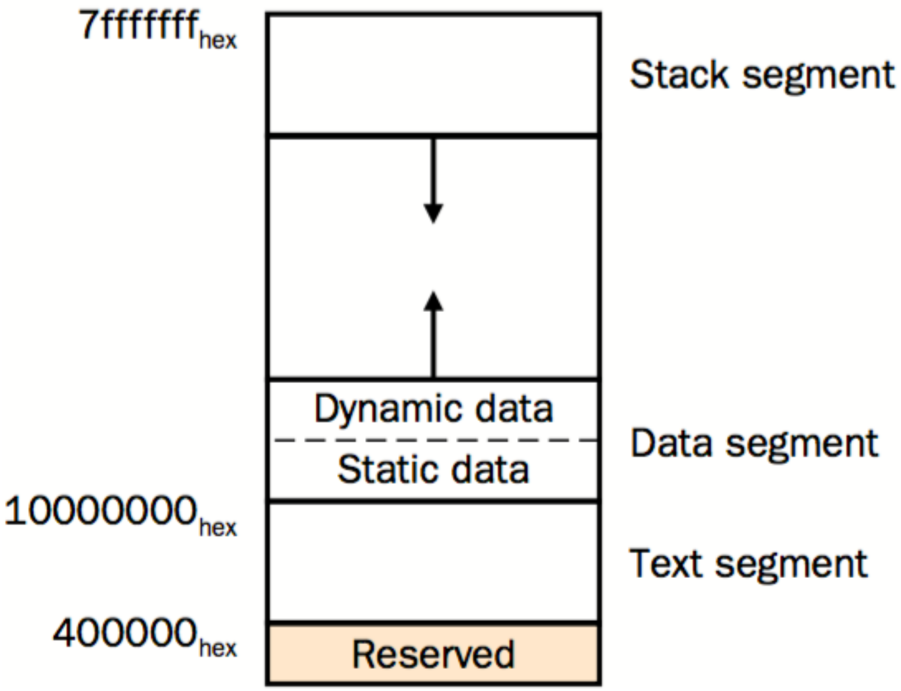


[Thain]



[Hennessy / Larus]

Memory Usage



Memory layout *convention* that is typically followed:

- [0x00000000 - 0x00400000] Reserved
- [0x00400000 - 0x10000000] Text Segment / Code (program's instructions)
- [0x10000000 - ..] Data Segments
  - begins with *Static* data
    - objects' size known to compiler
    - lifetime (interval in which program can access it) is program's entire execution
    - *Example:* constants
  - then *Dynamic* data / Heap
    - allocated as the program executes
    - in C, `malloc` library routine finds and returns a new-block of memory; in Java, the `new` keyword
    - grows upward toward stack (to higher addresses)
- [.. - 0x7fffffff] Stack Segment
  - grows downward (to lower addresses) towards heap as program pushes values onto stack
- Heap and stack begin as far apart as possible, but can grow to use a program's entire address space.

Procedure Call Convention

MIPS has 32 general purpose registers (numbered 0-31):

- 0 $zero $0 - hardwired with constant value 0
- 1 $at - reserved, should not be used by programmer
- 2-3 $v0 $v1 - convention: return values from functions
- 4-7 $a0 ... $a3 - pass first four args to routines; remaining args are passed on stack
- 8-15 $t0 ... $t7 - caller-save registers; not preserved across function calls
- 16-23 $s0 ... $s7 - callee-save registers
- 24-25 $t8 $t9 - caller-save
- 26-27 $k0 $k1 - reserved
- 28 $gp - points to static data segment "global pointer"; usually contains 0x10008000
- 29 $sp - stack pointer
- 30 $fp - frame pointer
- 31 $ra - return address from a procedure call

MIPS Commands to Know

- `lw rt, address`
  load the word from specified address into register rt
- `sw rt, address`
  store word in register rt into memory at specified address
- `addiu rt, rs, imm`
  imm is shorthand for "immediate"
  rt ← rs + imm
- `subu rd, rs, rt`
  rd ← rs – rt
- `jal target`
  unconditionally jump to instruction at label target
- `jr rs`
  unconditionally jump to instruction whose address is in register rs
- `mul rdest, rsrc1, src2`
  src2 is either a register or immediate value
- `move rdest, rsrc`
- `bgtz rs, label`
  branch to label if register rs is greater than zero
- `li rdesr, imm`
  load immediate

Examples in MARS 4.5
(MIPS Assembler and Runtime Simulator)

`add.asm`

```
main:
      addiu $t0, $zero, 5
      addiu $t1, $zero, 7
      addu  $t2, $t0, $t1
```

Examples in MARS 4.5
(MIPS Assembler and Runtime Simulator)

swap.asm

```
        .data
        .word 7
        .word 3

        .text
        .globl main
main:
        lui $s0, 0x1001     #load upper part of register s0 with 0x1001, so s0 = 0x10010000
        lw $s1, 0($s0)      #load s1 with the contents of memory address 0x10010000 = 7
        lw $s2, 4($s0)      #load s2 with the contents of memory address 0x10010004 = 3
        sw $s2, 0($s0)      #store contents of s2 into memory address 0x10010000
        sw $s1, 4($s0)      #store contents of s1 into memory address 0x10010004
        jr $ra
```

Procedure Call Convention

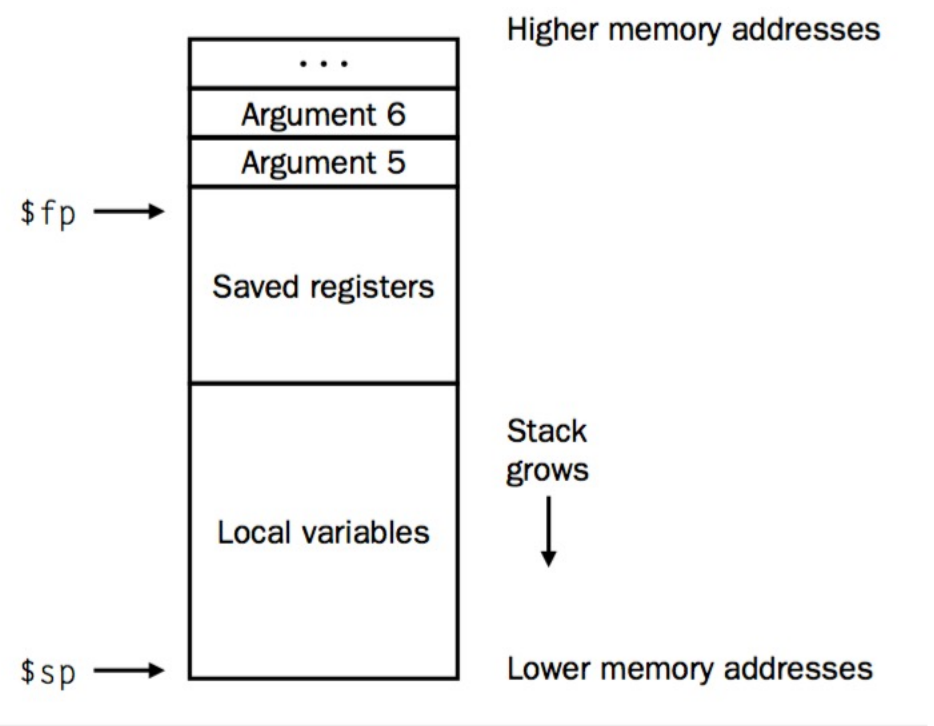
What *caller* does to execute a procedure call:

1. Pass arguments
   - first four passed in registers ($a0 .. $a3)
   - remaining args pushed on stack
2. Save *caller-save* registers
3. Execute `jal` instruction, which automatically:
   - jumps to *callee*'s first instruction
   - saves return address in $ra

Procedure Call Convention

What *callee* does before called subroutine begins execution:

1. Save callee-save registers in the frame ($s0 .. $s7) if going to alter:
   - $fp - saved by every procedure that allocates a new stack frame
   - $ra - only needs to be saved if *callee* itself makes a call
2. Establish frame pointer $fp
   - $fp = $sp + framesize - 4



Procedure Call Convention

How *callee* returns execution back to the *caller*:

- if *callee* is returning a value, place it in $v0
- restore all *callee-saved* registers that were saved upon procedure entry
- “pop the stack”

```
$sp = $sp + framesize
$sp = $fp + 4
```

- return by jumping to address in $ra

<u>Calling convention requires space in stack frame for:</u>

- 4 arg registers ($a0 .. $a3)
- return address ($ra)
  - (padded to <u>double word boundary</u>)

1 word = 4 bytes (32 bits)

- 4 words + 1 word pad + 1 word $ra = 24 bytes *(minimum frame size)*
- to store a word in byte-addressable memory, must break 32-bit quantity into 4 bytes:
  - *Example:* `0x01ab23cd`
  - *Big-endian:* `0x01 0xab 0x23 0xcd`
  - *Little-endian:* `0xcd 0x23 0xab 0x01`
- to make hardware simpler, words are stored at word-aligned addresses
  - $sp is also kept double word aligned
  - local data section is also double word aligned

Proposed Convention for all Procedure Calls

```
8($fp)
4($fp)
---------------------
0($fp)       Arg4
-4           Arg3
-8           Arg2
-12          Arg1
        <---------------------padding here so that return addr is double word aligned?
-16          return addr                  --|
-20          frame pt                       |--- saved registers
...          other callee saved registers --|

...          local variables and outgoing args
$sp     <---------------------double word aligned
----------------------
```

Examples

Factorial (Page A-26)

C routine:

```
main() {
    printf("The factorial of 10 is %d\n", fact(10));
}
int fact(int n) {
    if (n < 1)
        return 1;
    else
        return n * fact(n-1);
}
```

- `factorial.asm` as defined won't work because `printf` is unresolved referenced
    - Modified code to use `syscall`, pg A-48
- Demo `factorial.asm` in MARS

After the Break: Practice!

PBL Activity

- Teams of 2 or 3.

Problem 1a: “Getting Your Feet Wet”

Translate this Ram23-like program to assembly. Load this into your MIPS simulator and make sure it executes.

```
integer x;
integer y;
integer z;
x = 5;
x = 2 * x;
y = 3 + x;
z = y - x;
```

Problem 1b: “Adding println”

Then add `println`.

```
integer x;
integer y;
integer z;
x = 5;
x = 2 * x;
y = 3 + x;
z = y - x;

println(z);
```

- Output answer using `println`? See `syscall` (Page A-28)

```
li $v0, 1
li $a0, 5
Syscall // prints 5
// system call for print_int is 1
```

Problem 2: “Loops”

```
integer i = 0;
integer result = 0;
for (i = 0; i < 20; i=i+2)
    result += i;
println(result);
```

- See next slide for branching hint…
- What are the MIPS opcode instructions to know?
- How to perform a branch *(if statements)?*
    - *See next slide*

Branching Idea

```
    evaluation code that puts a value into a register

    bgtz FalseLabel

    code to run if branch is true
    jump instruction to DoneLabel

FalseLabel:
    code to run if branch is false

DoneLabel:
    rest of program
```

Problem 3: “A method call”

- Be especially comfortable with slides 21-25

```
main {
    integer a;
    a = 4;
    println(helper(a));
}
public static integer helper(integer a) {
    integer b;
    b = a * 4;
    return b;
}
```

Problem 4: “Factorial”

- Create <u>Factorial</u> and compute `factorial(10)`
- Be comfortable with Section A.6 before starting this problem.
- See starter code beginning on Page A-27.

More Practice? Problem 5: “Fibonacci”

- Create Fibonacci and compute `fib(5)`