CMSC 611: Advanced Computer Architecture

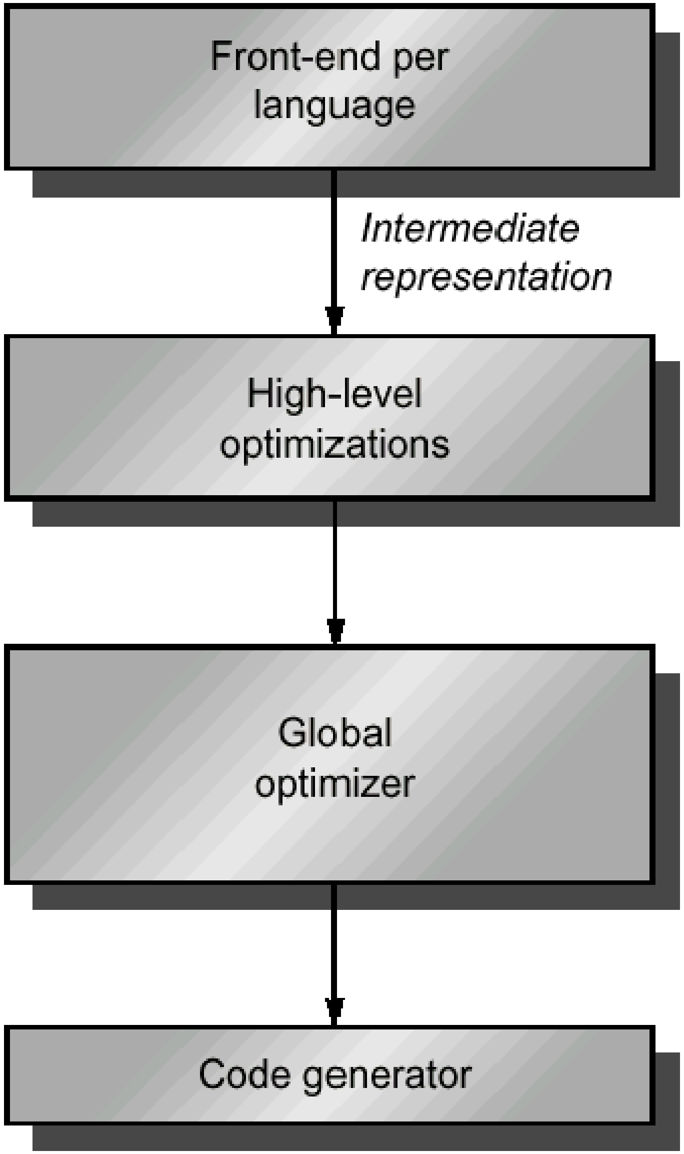
Compilers & ISA

Some material adapted from Mohamed Younis, UMBC CMSC 611 Spr 2003 course slides
Some material adapted from Hennessy & Patterson / © 2003 Elsevier Science

Lecture Overview

- Last week
  - Type and size of operands
    - (common data types, effect of operand size on complexity)
  - Encoding the instruction set
    - (Fixed, variable and hybrid encoding, stored program)
- This week
  - The role of the compiler
  - Effect of ISA on Compiler Complexity
  - Pipeline performance
  - Pipeline hazards

Typical Compilation



Dependencies		Function
Language dependent; machine independent	Front-end per language	Transform language to common intermediate form
	Intermediate representation	
Somewhat language dependent, largely machine independent	High-level optimizations	For example, procedure inlining and loop transformations
Small language dependencies; machine dependencies slight (e.g., register counts/types)	Global optimizer	Including global and local optimizations + register allocation
Highly machine dependent; language independent	Code generator	Detailed instruction selection and machine-dependent optimizations; may include

Compiling Array Indexing

Let's assume that A is an array of word-length integers and that the compiler has associated the variables g, h and i with the registers $s1, $s2 and $s4. Let's assume that the starting address, or base address, of the array is in $s3. The following is a possible compilation of a segment of a C program to MIPS assembly instructions:

g = h + A[i];

First convert word-index to byte-index:

```
add     $t1, $s4, $s4     # Temp reg $t1 = 2 * i
add     $t1, $t1, $t1     # Temp reg $t1 = 4 * i
```

To get the address of *A[i]*, we need to add *$t1* to the base of *A* in *$s3*:

```
add     $t1, $t1, $s3     # $t1 = address of A[i] (4 * i + $s3)
```

Now we can use that address to load A[i] into a temporary register:

```
lw      $t0, 0($t1)       # Temporary register $t0 gets A[i]
```

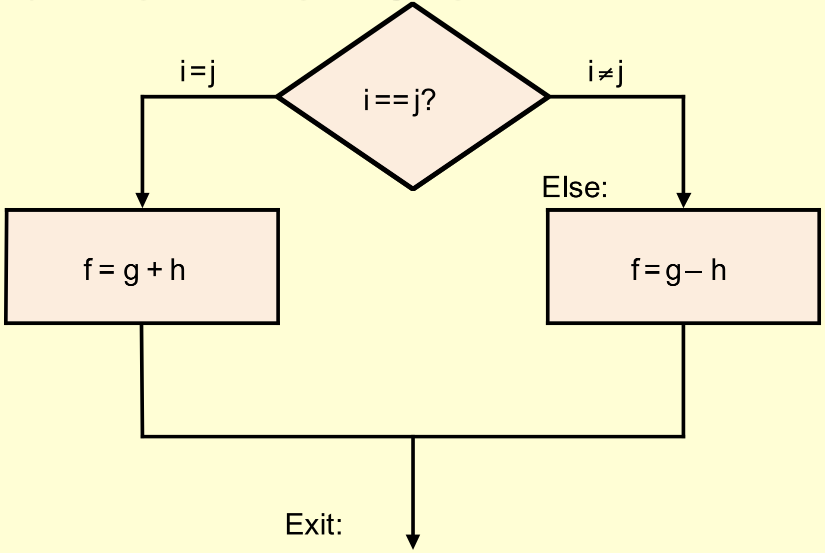
Finally add *A[i]* to *h* and place the sum in *g*:

```
add     $s1, $s2, $t0     # g = h + A[i]
```

Compiling if-then-else

Assuming the five variables f, g, h, i, and j correspond to the five registers $s0 through $s4, what is the compiled MIPS code for the following C if statement:

if (i == j) f = g + h; else f = g - h;



MIPS:

```
        bne     $s3, $s4, Else        # go to Else if i ≠ j
        add     $s0, $s1, $s2         # f = g + h (skipped if i ≠ j)
        j       Exit
Else:   sub     $s0, $s1, $s2         # f = g - h (skipped if i = j)
Exit:
```

Compiling a while Loop

Assume that i, j and k correspond to $s3 through $s5, and the base of the array "save" is in $s6. what is the compiled MIPS code for the following C segment:

```
while (save[i] == k)   i = i + j;
```

MIPS:

The first step is to load save[i] into a temporary register

```
Loop:   add     $t1, $s3, $s3              # Temp reg $t1 = 2 * i
        add     $t1, $t1, $t1              # Temp reg $t1 = 4 * i
        add     $t1, $t1, $s6              # $t1 = address of save[i]
        lw      $t0, 0($t1)                # Temp reg $t0 = save[i]
```

The next instruction performs the loop test, exiting if save[i] ≠ k

```
        bne     $t0, $s5, Exit             # go to Exit if save[i] ≠ k
```

The next instruction add j to i:

```
         add    $s3, $s3, $s4              # i =  i + j
```

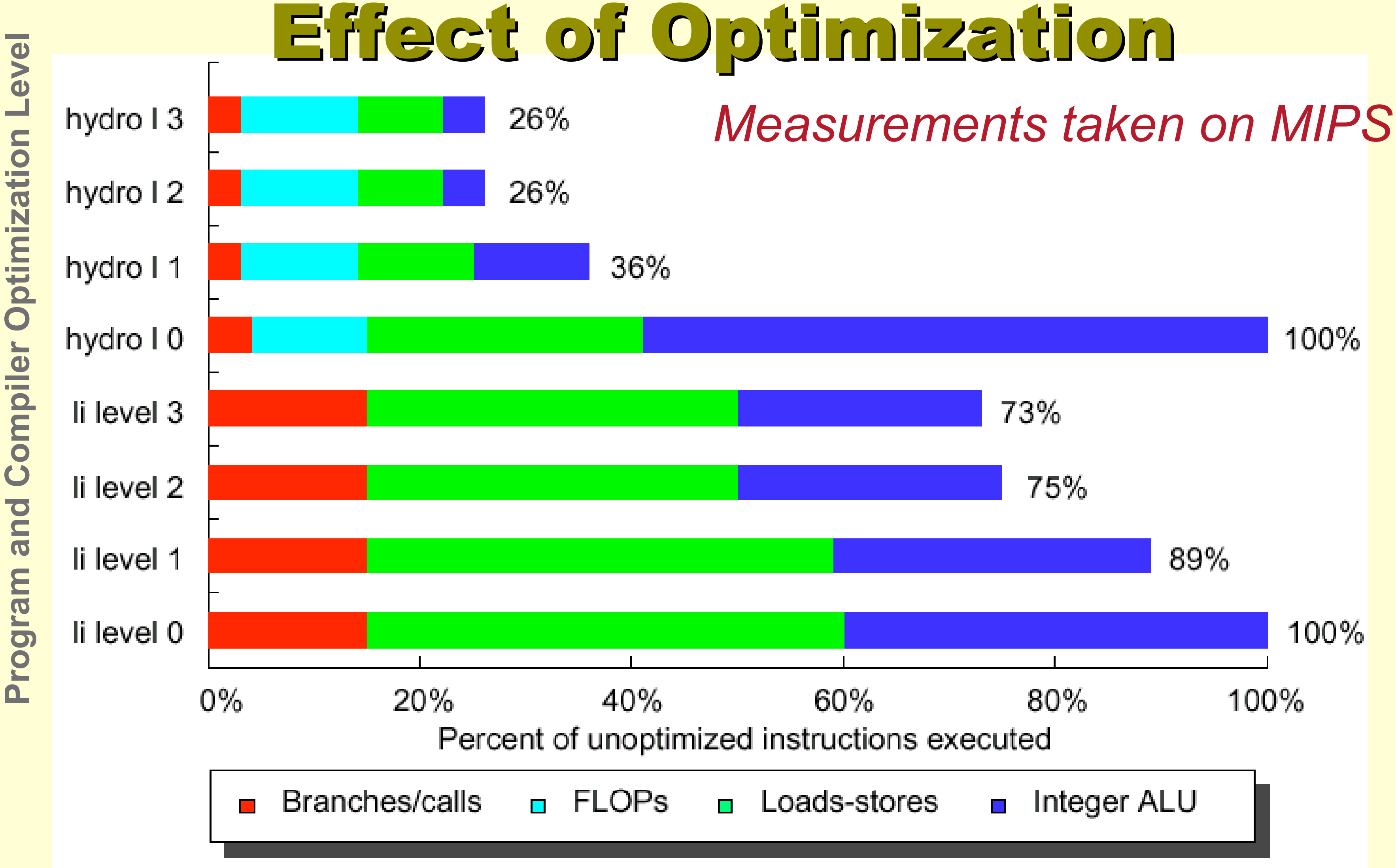
Finally reaching the loop end

```
        j       Loop                  # go back to the beginning of loop
Exit:
```

Major Types of Optimization

Optimization Name	Explanation	Frequency
High –level	*At or near source level; machine indep.*	
Procedure integration	Replace procedure call by procedure body	N.M
Local	*Within straight line code*	
Common sub- expression elimination	Replace two instances of the same computation by single copy	18%
Constant propagation	Replace all instances of a variable that is assigned a constant with the constant	22%
Stack height reduction	Rearrange expression tree to minimize resources needed for expression evaluation	N.M
Global	*Across a branch*	
Global common sub expression elimination	Same as local, but this version crosses branches	13%
Copy propagation	Replace all instances of a variable A that has been assigned X (i.e., A = X) with X	11%
Code motion	Remove code from a loop that computes same value each iteration of the loop	16%
Induction variable elimination	Simplify/eliminate array –addressing calculations within loops	2%
Machine-dependant	*Depends on machine knowledge*	
Strength reduction	Many examples, such as replace multiply by a constant with adds and shifts	N.M
Pipeline Scheduling	Reorder instructions to improve pipeline performance	N.M.

Effect of Optimization



Measurements taken on MIPS

Level 0: non-optimized code

Level 1: local optimization

Level 2: global optimization, s/w pipelining

Level 3: adds procedure integration

Multimedia Instructions

- Small vector processing targeting multimedia
  - Intel’s MMX, PowerPC AltiVec, Sparc VIS, MIPS MDMX
  - N 1/Nth-word vectors packed into one register
  - Same operations performed on all N vectors
- Plus
  - Little additional ALU hardware
  - Utilize under-used hardware resources
- Minus
  - Extra pack & unpack if data isn’t already arranged perfectly
  - Limited vector sizes, difficult to compile for general code
- Result
  - Mostly used in hand-coded libraries
- Compare to general vector processing
  - Hide memory latency in vector access
  - Strided processing, gather/scatter addressing

Effect of Compilers on ISA

- Promote regularity
  - Limit # register formats and variability of operands
  - Orthogonality in operations, registers & addressing
- Provide primitives, not solutions
  - Common features over specific language features
  - Special-purpose instructions often unusable (except through hand-assembly-coded libraries)
- Simplify trade-offs among alternatives
  - Simplify the analysis of special features such as cache and pipeline
  - Allow simultaneous activities to promote optimization
- Favor static binding

Starting a Program

Object files for Unix typically contains:

Header: size & position of components

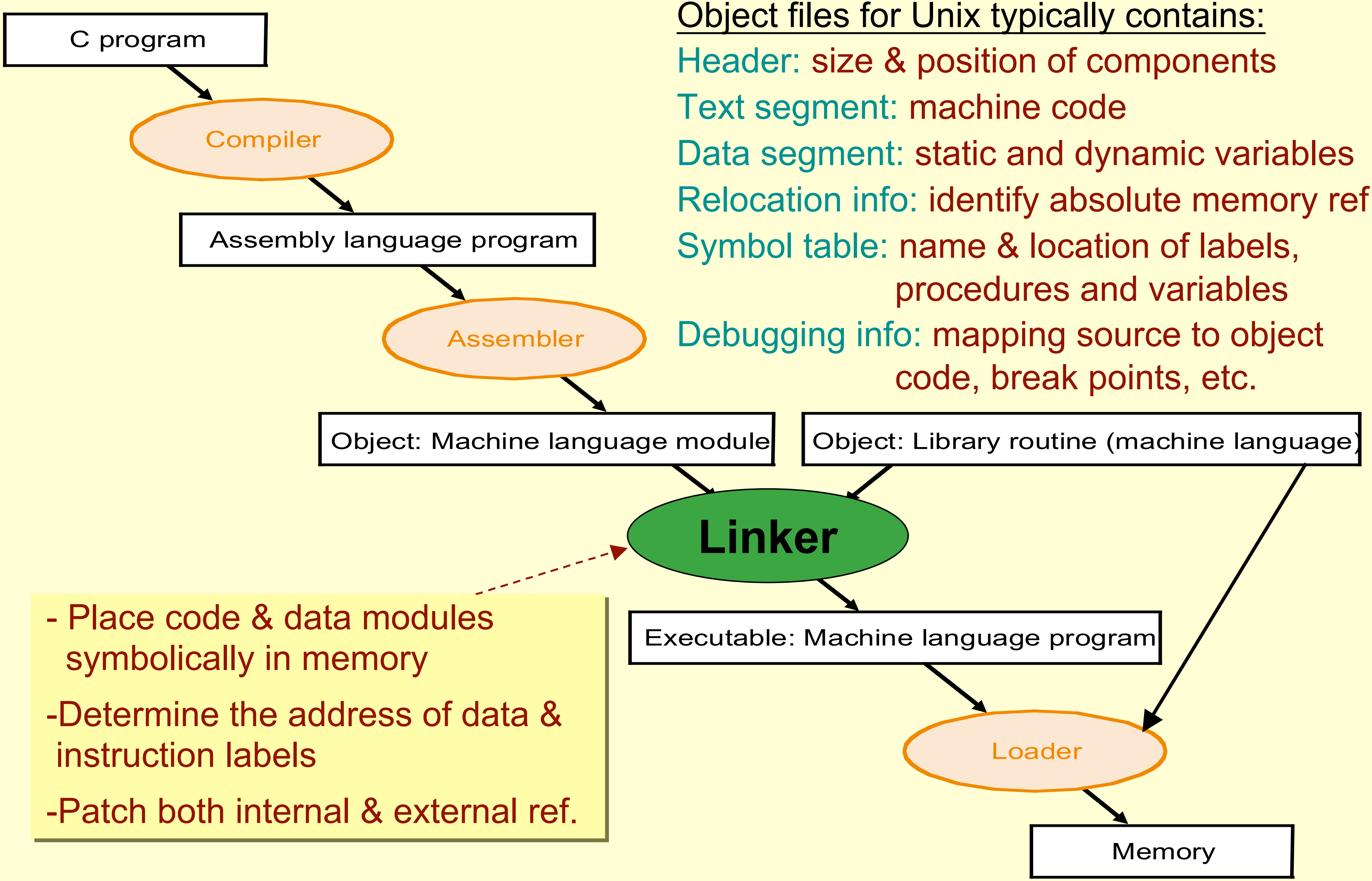
Text segment: machine code

Data segment: static and dynamic variables

Relocation info: identify absolute memory ref.

Symbol table: name & location of labels, procedures and variables

Debugging info: mapping source to object code, break points, etc.



C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory

- Place code & data modules symbolically in memory

-Determine the address of data & instruction labels

-Patch both internal & external ref.

Linking Object Files

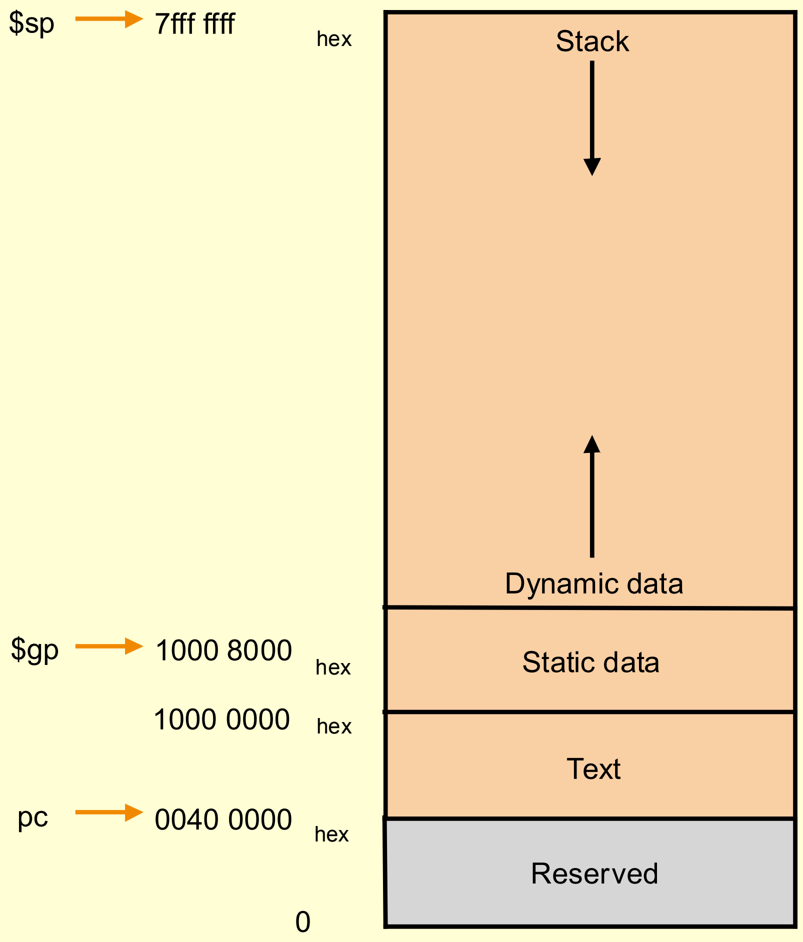
Object file header			
	Name	Procedure A	
	Text size	100_{hex}	
	Data size	20_{hex}	
Text segment	Address	Instruction	
	0	lw \$a0, 0(\$gp)	
	4	jal 0	
	…	…	
Data segment	0	(X)	
	….	…	
Relocation Info	Address	Instruction type	Dependency
	0	lw	X
	4	jal	B
Symbol table	Label	Address	
	X	-	
	B	-	

Object file header			
	Name	Procedure B	
	Text size	200_{hex}	
	Data size	30_{hex}	
Text segment	Address	Instruction	
	0	lw \$a0, 0(\$gp)	
	4	jal 0	
	…	…	
Data segment	0	(Y)	
	….	…	
Relocation Info	Address	Instruction type	Dependency
	0	lw	Y
	4	jal	A
Symbol table	Label	Address	
	Y	-	
	A	-	

Executable file header		
	Text size	300_{hex}
	Data size	50_{hex}
Text segment	Address	Instruction
	$0040\ 0000_{hex}$	lw \$a0, 8000_{hex}(\$gp)
	$0040\ 0004_{hex}$	jal $40\ 0100_{hex}$
	…	…
	$0040\ 0100_{hex}$	lw \$a1, 8020_{hex}(\$gp)
	$0040\ 0104_{hex}$	jal $40\ 0000_{hex}$
	…	…
Data segment	Address	
	$1000\ 0000_{hex}$	(X)
	…	…
	$1000\ 0020_{hex}$	(Y)
	…	…

Assuming the value in \$gp is $1000\ 8000_{hex}$

Loading Executable Program



- To load an executable, the operating system follows these steps:
  1. Read the executable file header to determine the size of text and data segments
  2. Create an address space large enough for the text and data
  3. Copy the instructions and data from the executable file into memory
  4. Copy the parameters (if any) to the main program onto the stack
  5. Initialize the machine registers and sets the stack pointer to the first free location
  6. Jump to a start-up routines that copies the parameters into the argument registers and calls the main routine of the program