ELT3047 Computer Architecture

Lecture 5: MIPS ISA (2)

Hoang Gia Hung
Faculty of Electronics and Telecommunications
University of Engineering and Technology, VNU Hanoi

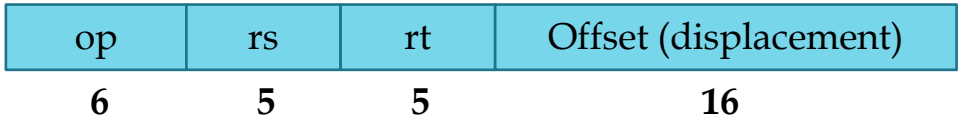
Last lecture review (1)

- Applications of design principles to MIPS ISA
  - Storage model, taking into account compiler consideration
  - Addressing modes
  - Instruction encoding
- MIPS instruction set examples
  - Arithmetic & logical instructions.

R-type

  - Data transfer & conditional branch instructions

I-type



  - Unconditional jump instructions

J-type

Last lecture review (2)

- MIPS instruction set examples (cont.)
  - If statement.
  - Loops
  - Inequalities
- **Today's lecture**: some more complex operations and compiling from HLL to MIPS assembly
  - Brief introduction to compiler technology.

Procedure/function call: overview

- Procedure/function call
  - **Spy analogy**: Leaves with a secret plan, acquires resources, performs the task, covers their tracks & returns to the point of origin with desired result.
  - One way to implement abstraction in software
- Procedure conventions:
  - **Caller**:
    - passes **arguments** to callee
    - jumps to callee
  - **Callee**:
    - **performs** the function
    - **returns** result to caller
    - **returns** to point of call
    - must **not** overwrite registers or memory needed by caller

Factorial C Code

```
int factorial(int n)
  {
    if (n < 1) return (1);
    else return n * factorial(n - 1);
  }
```

Compiling a procedure call in MIPS

- Steps required & architectural support in MIPS

Step	Description	MIPS implementation
1	Place parameters in registers	`$a0 - $a3`
2	Transfer control to procedure & save return address	`jal, $ra`
3	Acquire storage for procedure	`$gp, $sp, $fp`
4	Perform procedure's operations	`$t0 – $t9, $s0 – $s7`
5	Place result in register for caller	`$v0, $v1`
6	Return to place of call	`jr $ra`

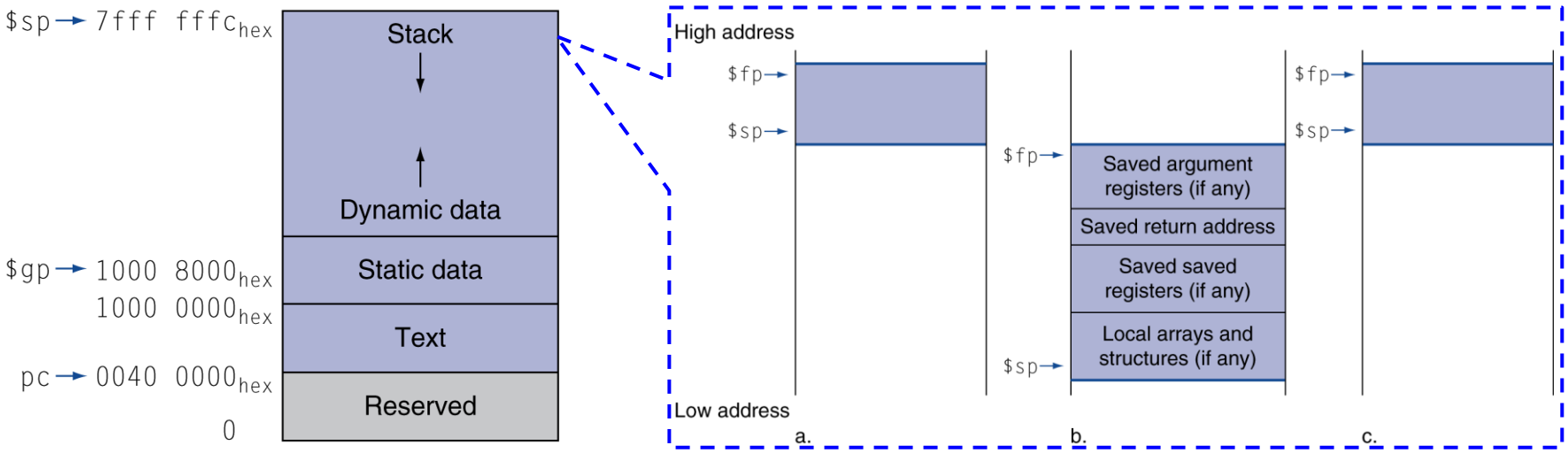
- Register usage:
  - Caller puts arguments in `$ a0– $ a3`, jumps to callee & simultaneously saves `$PC + 4` in register `$ra` via the instruction `jal &(callee)`
  - Some temporary registers (e.g. `$s0` – `$s7`) must be restored to the values they hold ***before*** the procedure was invoked → spill registers to **stack**.
  - When the callee finishes calculations, it places the results in `$v0` & `$v1`, and returns control to the caller using `jr $ra`.

MIPS call convention: registers

- MIPS register file partition:
  - `$a0-$a3`: arguments (reg's 4 – 7)
  - `$v0, $v1`: result values (reg's 2 and 3)
  - `$t0-$t9`: temporaries
  - `$s0-$s7`: variables
  - `$gp`: global pointer for static data (reg 28)
  - `$sp`: stack pointer (reg 29)
  - `$fp`: frame pointer (reg 30)
  - `$ra`: return address (reg 31)
- Register preserving convention:
  - to reduce **register spilling**

Preserved (callee-saved/non volatile)	Not Preserved (caller-saved/volatile)
Saved registers: `$s0-$s7`	Temporary registers: `$t0-$t9`
Stack Pointer: `$sp`	Argument registers: `$a0-$a3`
Return Address Register: `$ra`	Return registers: `$v0-$v1`

MIPS call convention: memory layout



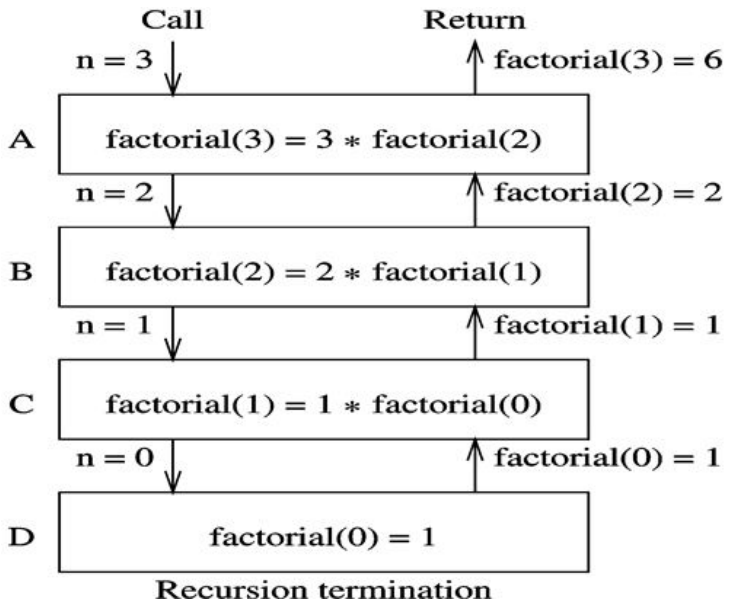
Stack = collection of stack frames (aka activation record), each frame corresponds to a call to a subroutine which has not yet terminated.

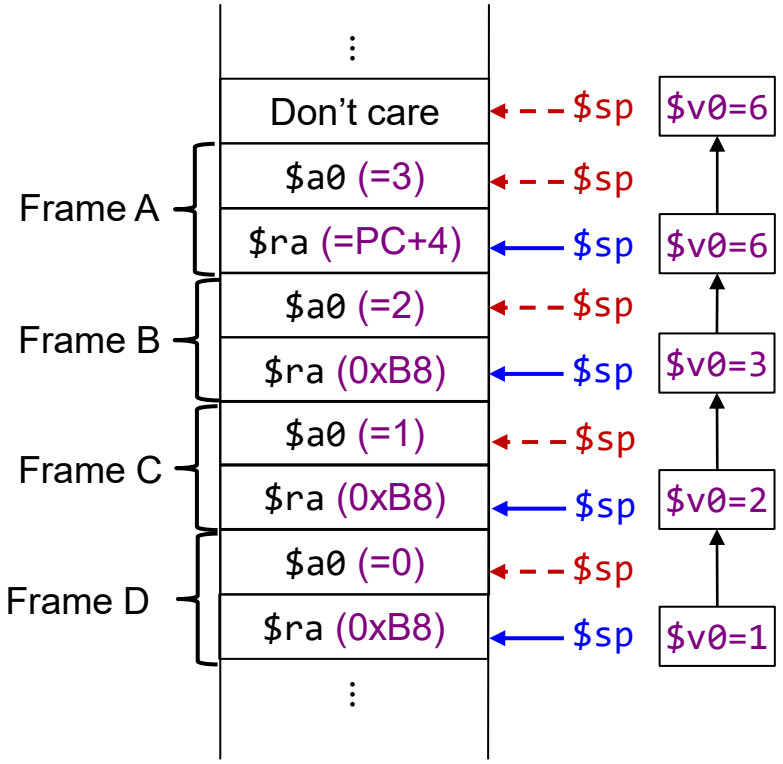
- Memory allocation for different types of data in HLL
  - Program code → Text segment
  - **Static/global** variables → $gp
  - **Automatic/local** variables → $sp, $fp (while managing complex local variables that do **not** fit in registers, such as ***local arrays*** or ***structures***).
  - **Dynamic data** structures (e.g. linked lists) → **heap** (grows ↑, stack grows ↓)

Example: factorial procedure

```
0x90 factorial: addi $sp, $sp, -8  # make room (2 words) for stack
0x94            sw   $a0, 4($sp)   # store $a0
0x98            sw   $ra, 0($sp)   # store $ra
0x9C            slti $t0, $a0, 1   # a < 1 ?
0xA0            beq  $t0, $0, else # no: go to else
0xA4            addi $v0, $0, 1    # yes: return 1
0xA8            addi $sp, $sp, 8   # restore $sp (pop 2 words)
0xAC            jr   $ra           # return
0xB0      else: addi $a0, $a0, -1  # n = n - 1
0xB4            jal  factorial     # recursive call
0xB8            lw   $ra, 0($sp)   # restore $ra
0xBC            lw   $a0, 4($sp)   # restore $a0
0xC0            addi $sp, $sp, 8   # restore $sp (pop 2 words)
0xC4            mul  $v0, $a0, $v0 # n * factorial(n-1)
0xC8            jr   $ra           # return
```

Stack layout during factorial execution

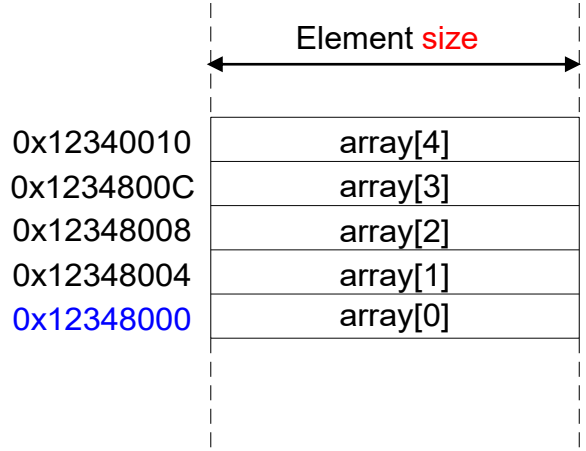




- No need for `$fp` in the example as the stack is adjusted only on entry and exit of the procedure → can be managed with only `$sp`.
- Stack frame (activation record) appears on the stack whether or not an explicit frame pointer is used.

Array and Loop

- Access large amounts of similar data (in memory)
  - Index vs Pointer

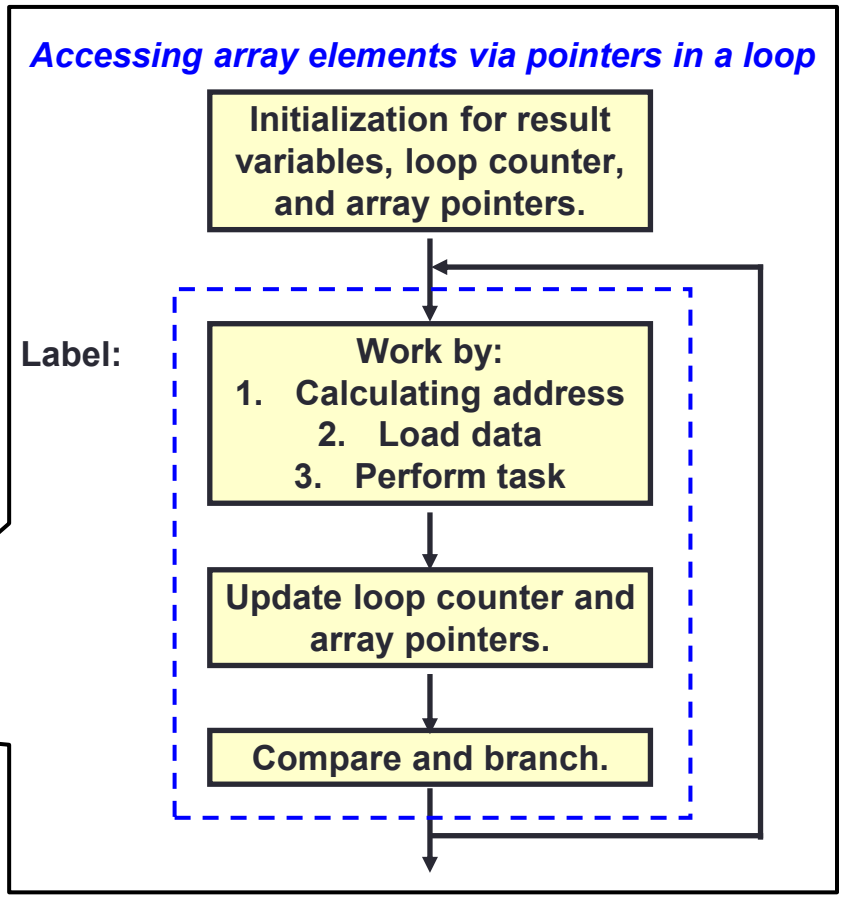


Array indexing involves

- Multiplying **index** by element size
- Adding to array base address

Pointers correspond directly to memory addresses

- can avoid indexing complexity



Array and Loop: example

- Count the number of zeros in an array `A`
  - `A` is word array with 40 elements
  - Address of `A[]` → `$t0`, result → `$t8`

Zero count C code

```
result = 0;

i = 0;

while ( i < 40 ) {
    if ( A[i] == 0 )
        result++;
    i++;

}
```

- Compiling: think about
  - How to perform the right comparison
  - How to translate `A[i]` correctly

Array and Loop: Version 1.0

Address of A[] ➔ $t0 Result ➔ $t8 i ➔ $t1	Comments

addi $t8, $zero, 0
addi $t1, $zero, 0
addi $t2, $zero, 40
loop: bge $t1, $t2, end
sll $t3, $t1, 2
add $t4, $t0, $t3
lw $t5, 0($t4)
bne $t5, $zero, skip
addi $t8, $t8, 1
skip: addi $t1, $t1, 1
j loop
end:
	# end point
	
	# i * 4
	# &A[i]
	# $t5 ← A[i]
	
	# result++
	# i++

Array and Loop: Version 2.0

Address of A[] ➔ $t0 Result ➔ $t8 &A[i] ➔ $t1	Comments

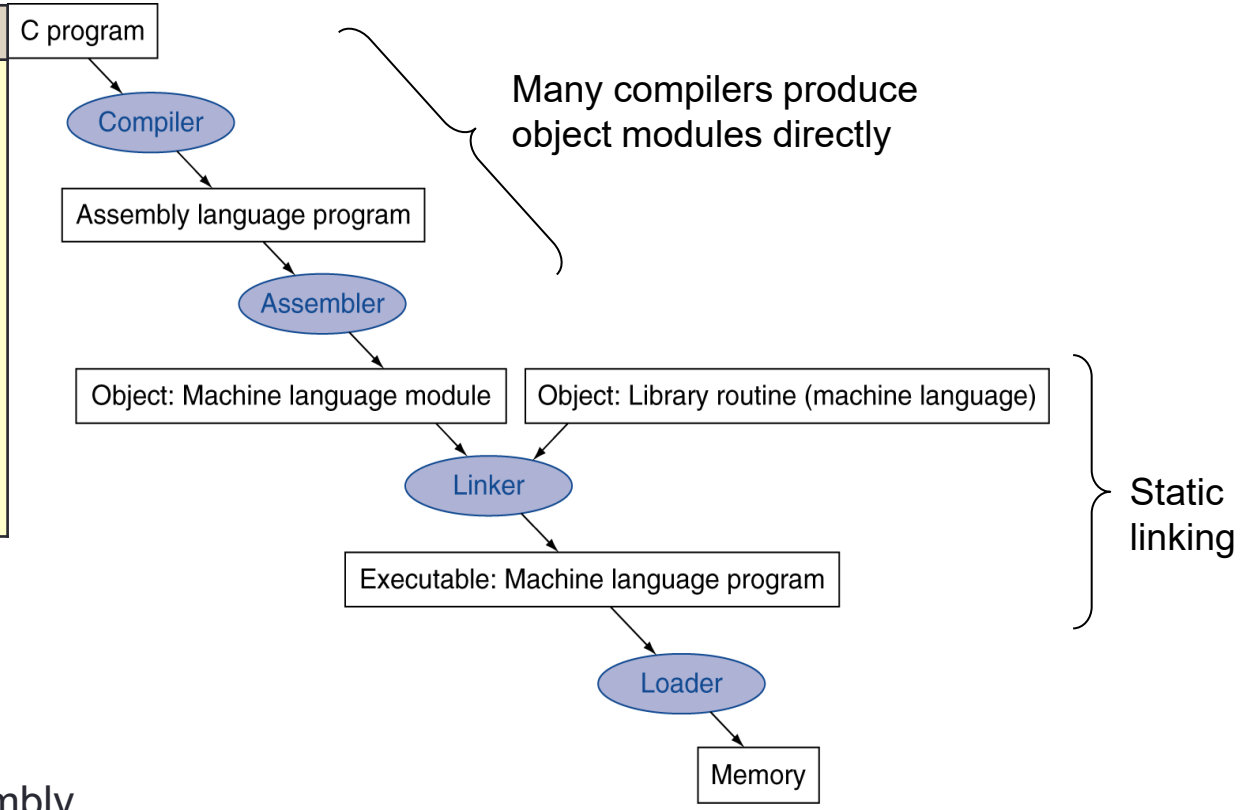
addi $t8, $zero, 0
addi $t1, $t0, 0
addi $t2, $t0, 160
loop: bge $t1, $t2, end
lw $t3, 0($t1)
bne $t3, $zero, skip
addi $t8, $t8, 1
skip: addi $t1, $t1, 4
j loop
end:
	# pointer to &A[current]
	# end point: &A[40]
	# comparing address!
	# $t3 ← A[i]
	
	# result++
	# move to next item

- Use of "pointers" can produce more efficient code!
  - Reduces the instructions executed per iteration by 2

Translating and Starting a Program

Example C code

```
int f, g, y;//global variables
int main(void)
{
  f = 2;
  g = 3;
  y = sum(f, g);
  return y;
}
int sum(int a, int b) {
  return (a + b);
}
```



HLL advantages over assembly

- Productivity (concise, readable, maintainable)
- Correctness (type checking, etc)
- Portability (run on different HW)

HLL disadvantages over assembly

- Efficiency?

Compiler: overview

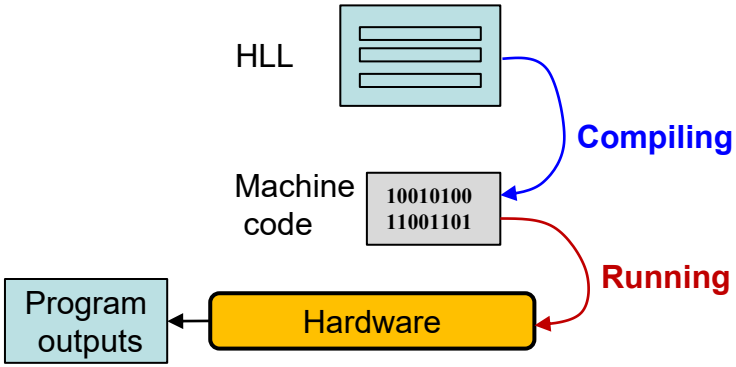
- Assembler (or compiler) translates program into machine instructions
  - Most assembler instructions represent machine instructions one-to-one
  - **Pseudo-instructions**: figments of the assembler's imagination, e.g.

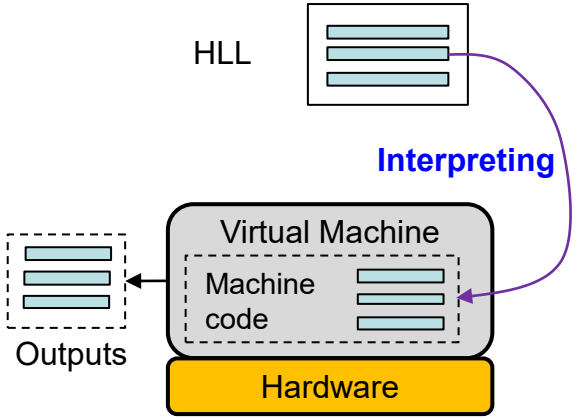
    `move $t0, $t1 → add $t0, $zero, $t1`

- Provides information for building a complete program from the pieces
  - **Header**: described contents of object module
  - **Text segment**: translated instructions
  - **Static data segment**: data allocated for the life of the program
  - **Relocation information**: for contents that depend on absolute location of loaded program
  - **Symbol table**: global definitions and external refs
  - **Debug info**: for associating with source code

Compiler: characteristics

- Compilation versus interpretation





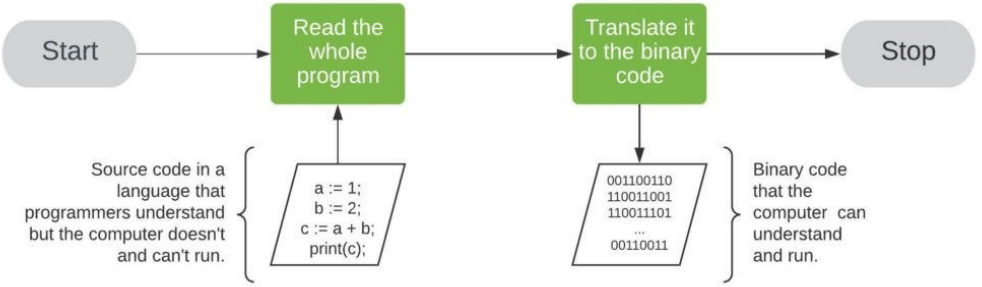
- Characteristics

	Compiler	Interpreter
How it converts the input?	Entire program at once	Read one instruction at a time
When is it needed?	Once, before the 1st run	Every time the program is run
Decision made at	Compile time	Run time
What it slows down	Program development	Program execution

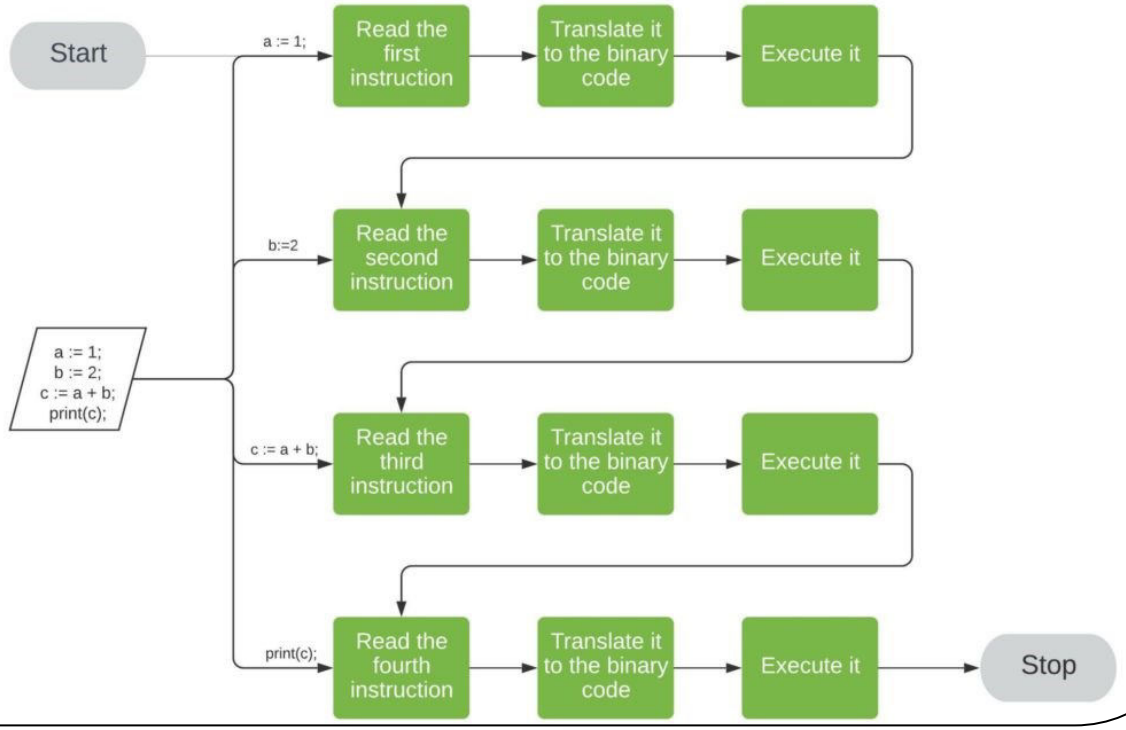
  - Some languages mix both concepts, e.g. Java.

Compilation vs interpretation illustration

Compilation

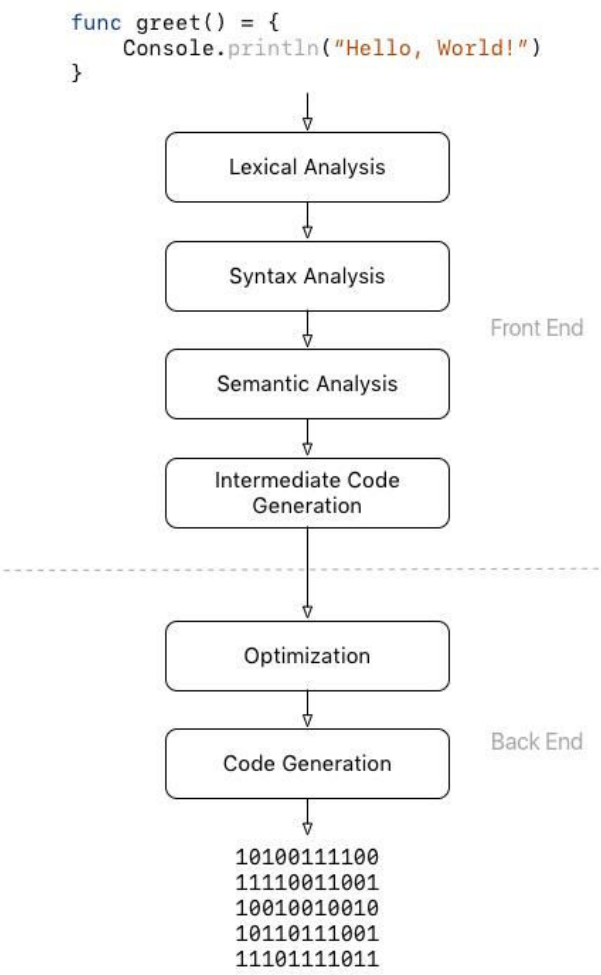


Interpretation



Anatomy of a compiler

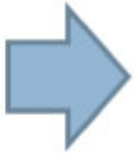
- **Frontend (analysis phase)**
  - Read source code text & break it up into meaningful elements (*lexeme*) & gen. tokens
    - ***token*** = {type, location} of a lexeme
  - Check correctness, report errors
    - e.g. “3x” is an illegal token in C
  - Translate to **intermediate representation**
    - **IR** = machine-independent language
    - e.g. *three-address code* (*3AC*)
- **Backend (synthesis)**
  - Optimize IR
    - reduces #operations to be executed
  - Translate IR to assembly & further optimize
    - take advantage of particular features of the ISA
    - e.g. `mult ← sll`



Front end stages: Lexical Analysis

- **Lexical Analysis** (**scanning**)
  - Source → list of tokens.

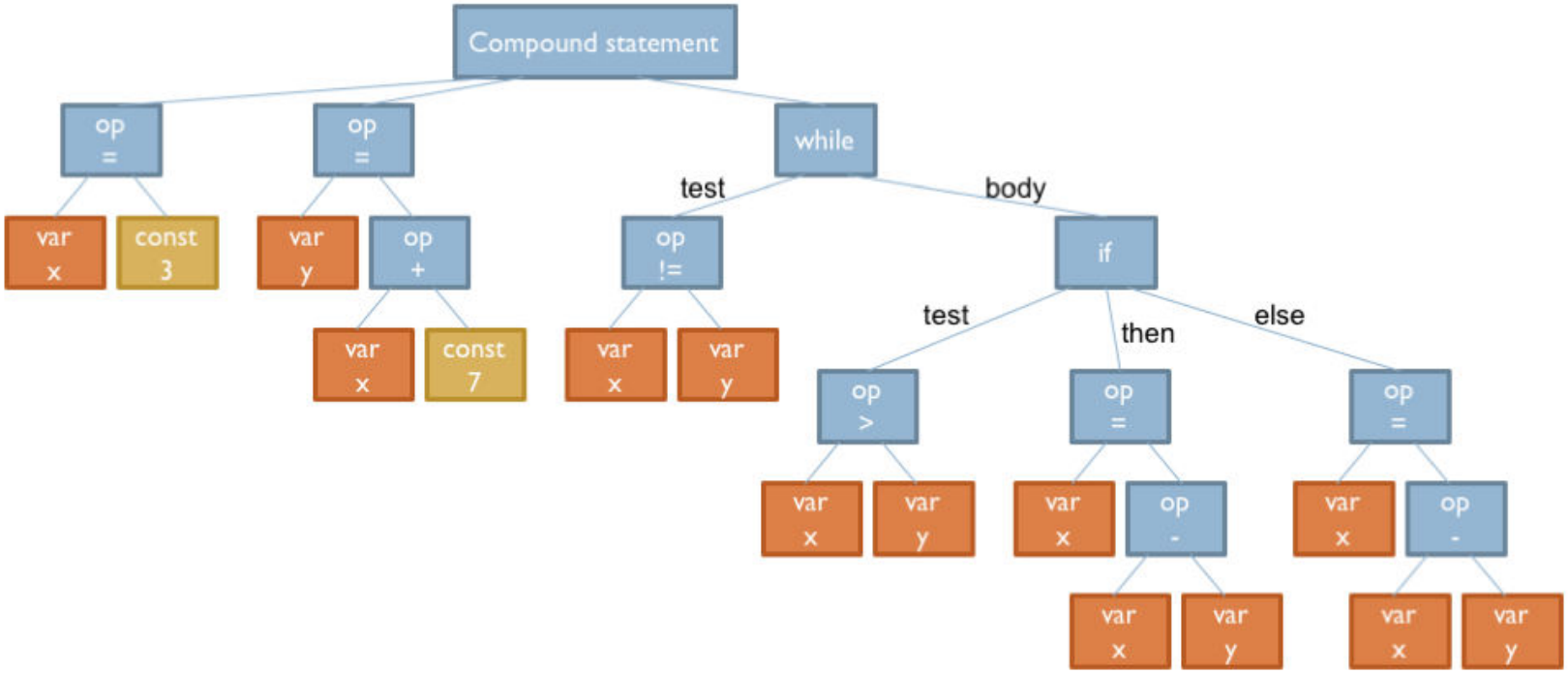
```
int x = 3;
int y = x + 7;
while (x != y) {
  if (x > y) {
    x = x - y;
  } else {
    y = y - x;
  }
}
```



```
("int", KEYWORD)
("x", IDENTIFIER)
("=", OPERATOR)
("3", INT_CONSTANT)
(";", SPECIAL_SYMBOL)
("int", KEYWORD)
("y", IDENTIFIER)
("=", OPERATOR)
("x", IDENTIFIER)
("+", OPERATOR)
("7", INT_CONSTANT)
(";", SPECIAL_SYMBOL)
("while", KEYWORD)
("(", SPECIAL_SYMBOL)
...
```

Front end stages: Syntax Analysis

- **Syntax Analysis** (**parsing**)
  - Tokens → syntax tree = syntactic structure of the original source code (text)



Front end stages: Semantic Analysis

- **Semantic Analysis**
  - Mainly type checking, i.e. if types of the operands are compatible with the requested operation.

Consider:

```
int x = "bananas";
```

Syntax OK
Semantically (meaning) WRONG

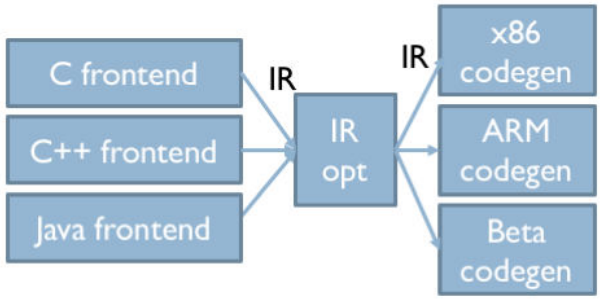
Symbol table

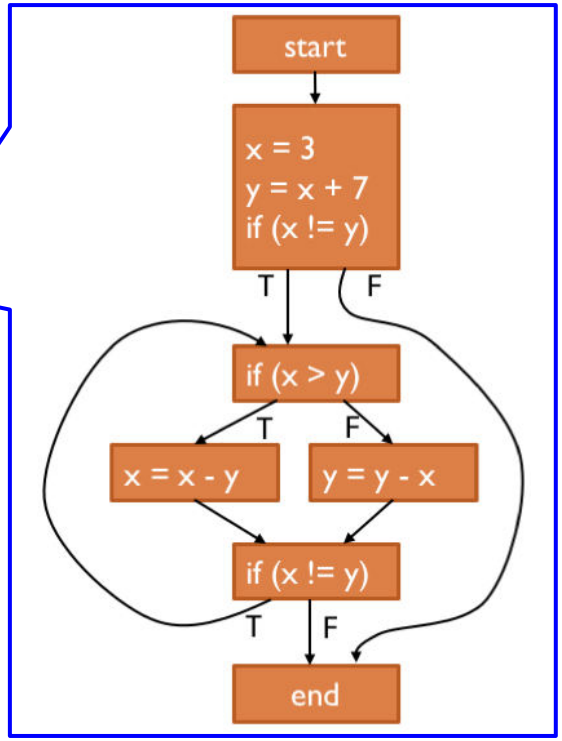
Var	Type
x	int

```
Line 1: error, invalid conversion from string constant to int
```

Front end stages: Intermediate representation (IR)

- **Internal** compiler language that is
  - Language-independent
  - Machine-independent
  - Easy to optimize
- Why yet another language?
  - Assembly does not have enough info to optimize it well
  - Enable modularity and reuse
- A common IR: **Control Flow Graph** (CFG)
  - **Nodes**: basic blocks = sequences of operations that are executed as a unit
  - **Edges**: branches connecting basic blocks





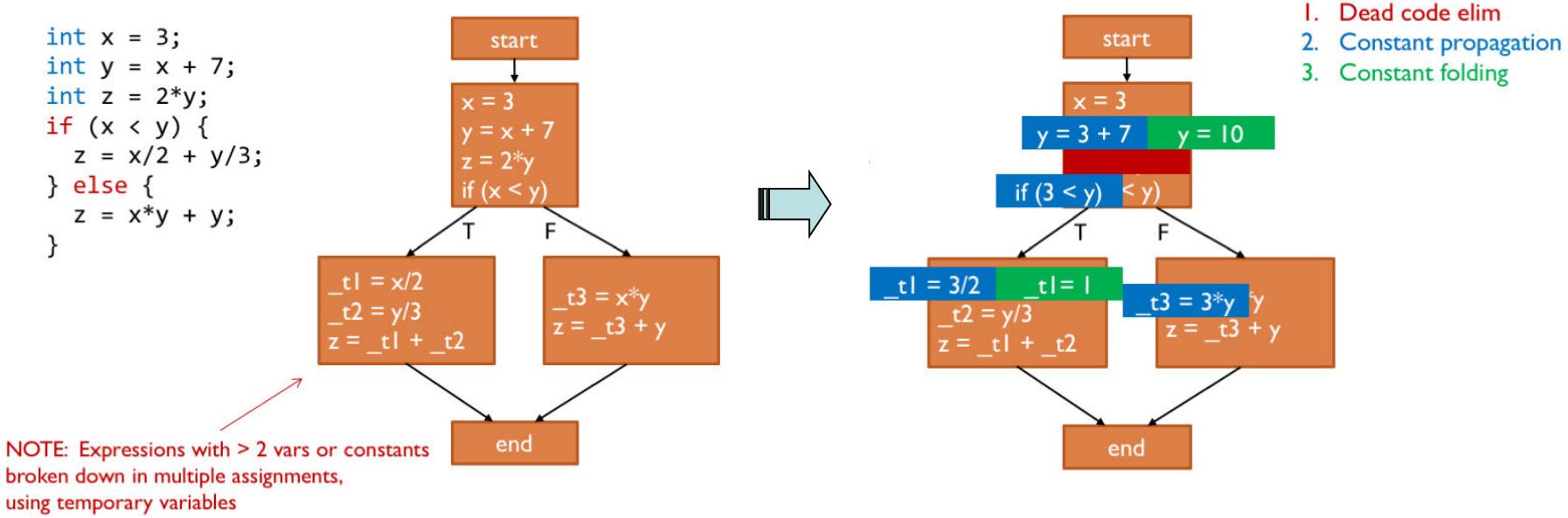
Backend stages: IR optimization

- Perform multiple **passes** over the CFG
  - A pass = a specific, simple optimization.
  - Repeatedly apply multiple passes until no further optimization can be found.
  - Combination of multiple simple optimizations = very complex optimizations.
- Typical optimizations:
  - **Dead code elimination**: eliminate assignments to variables that are never used and basic blocks that are never reached.
  - **Constant propagation**: identify variables that have a constant value & substitute that constant in place of references to the variable.
  - **Constant folding**: compute expressions with all constant operands.
  - **Example**: optimize

```
int x = 3;
int y = x + 7;
int z = 2*y;
if (x < y) {
  z = x/2 + y/3;
} else {
  z = x*y + y;
}
```

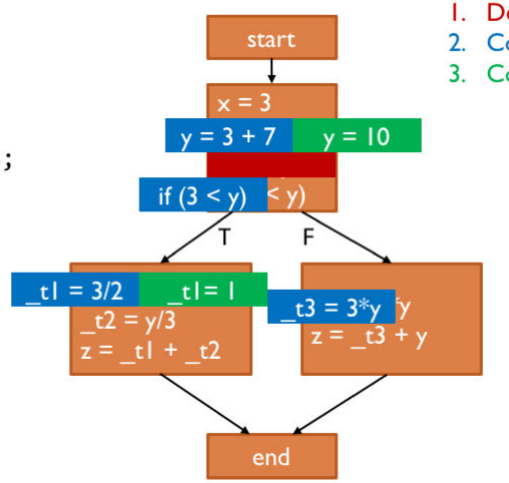
IR optimization example: 1st batch of passes

- First 3 passes:
  1. **Dead code elimination**: remove the assignment to **z** in the first basic block.
  2. **Constant propagation**: replace all references to **x** with the constant 3.
  3. **Constant folding**: compute constant expressions: `y=3+7=10` & `_t1=3/2=1`

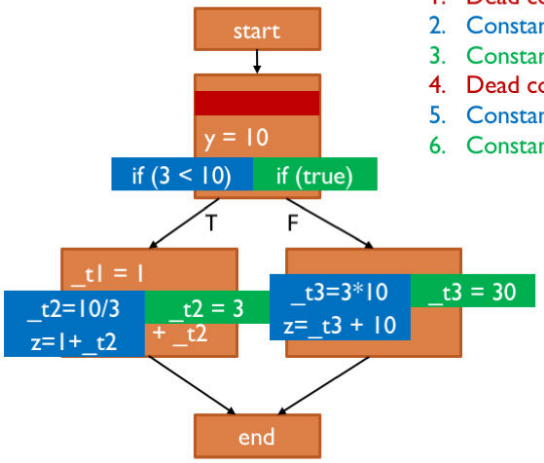


IR optimization example: subsequent batches of passes

```
int x = 3;
int y = x + 7;
int z = 2*y;
if (x < y) {
  z = x/2 + y/3;
} else {
  z = x*y + y;
}
```

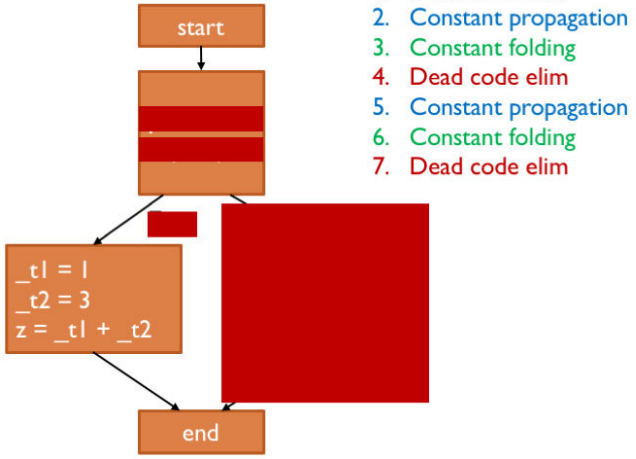


1. Dead code elim
2. Constant propagation
3. Constant folding



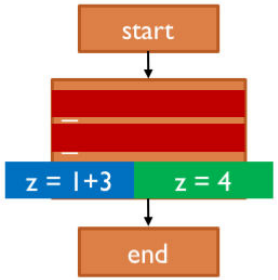
1. Dead code elim
2. Constant propagation
3. Constant folding
4. Dead code elim
5. Constant propagation
6. Constant folding

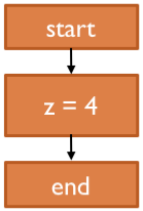
Repetition of simple optimizations on CFG = Very powerful optimizations



1. Dead code elim
2. Constant propagation
3. Constant folding
4. Dead code elim
5. Constant propagation
6. Constant folding
7. Dead code elim

1. Dead code elim
2. Constant propagation
3. Constant folding
4. Dead code elim
5. Constant propagation
6. Constant folding
7. Dead code elim
8. Constant propagation
9. Constant folding
10. Dead code elim





No further optimization found

Backend stages: Code generation

- Translate IR to assembly
  - Map variables to registers (**register allocation**)
    - Code generator assigns each variable a dedicated register.
    - If #variables > #registers, map some less frequently used variables to Mem and load/store them when needed.
  - Translate each assignment to instructions
    - Some assignments requires > 1 instructions.
  - Emit each basic block
    - Codes + appropriate labels and branches.
  - Reorder basic block code wherever possible
    - to eliminate superfluous jumps.
  - Perform ISA- and CPU-specific optimizations
    - e.g. reorder instructions to improve performance.

Compiler output: an object file example

Object file

Object file header	Text Size	Data Size
	0x34 (52 bytes)	0xC (12 bytes)

Text segment	Address	Instruction
	0x00400000	0x23BDFFFC
	0x00400004	0xAFBF0000
	0x00400008	0x20040002
	0x0040000C	0xAF848000
	0x00400010	0x20050003
	0x00400014	0xAF858004
	0x00400018	0x0C10000B
	0x0040001C	0xAF828008
	0x00400020	0x8FBF0000
	0x00400024	0x23BD0004
	0x00400028	0x03E00008
	0x0040002C	0x00851020
	0x00400030	0x03E00008

Data segment	Address	Data
	0x10000000	f
	0x10000004	g
	0x10000008	y

Compiled (assembly) code

```
.data
f:
g:
y:
.text
main: addi $sp, $sp, -4 #init stack
      sw   $ra, 0($sp)  # push $ra
      addi $a0, $0, 2   # $a0 = 2
      sw   $a0, f       # f = 2
      addi $a1, $0, 3   # $a1 = 3
      sw   $a1, g       # g = 3
      jal  sum          # call sum
      sw   $v0, y       # y = sum()
      lw   $ra, 0($sp)  # old $ra
      addi $sp, $sp, 4  # pop stack
      jr   $ra          # return
sum:  add  $v0, $a0, $a1
      jr   $ra
```

Symbol	Address
f	0x10000000
g	0x10000004
y	0x10000008
main	0x00400000
sum	0x0040002C

Linker

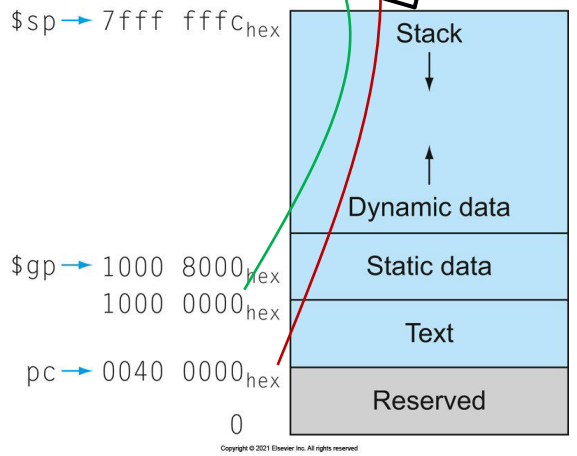
- Produces an executable image
  1. Merges segments (i.e. “stitches” standard library routines together)
  2. Resolve labels (determine their addresses) through relocation information and symbol table in each object module
  3. Patch location-dependent and external refs
- Executable file has the same format as an object file, except that it contains no unresolved references.
  - Some location dependencies might be fixed by relocating loader, but with virtual memory, no need to do this
  - Program can be loaded into absolute location in virtual memory space
- **Dynamic Linking**
  - **Static** linking: library routines is part of the executable code → will not be updated with new versions. Furthermore, it loads all routines in the library even if those are not executed (**image bloat**).
  - **Dynamically** linked libraries (**DLL**s): only link/load library procedure when it is called → keep location of nonlocal procedures and their names.

Linking object files example

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 information	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	sw $a1, 0($gp)	
	4	jal 0	
	...	...	
Data segment	0	(Y)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	sw	Y
	4	jal	A
Symbol table	Label	Address	
	Y	–	
	A	–	

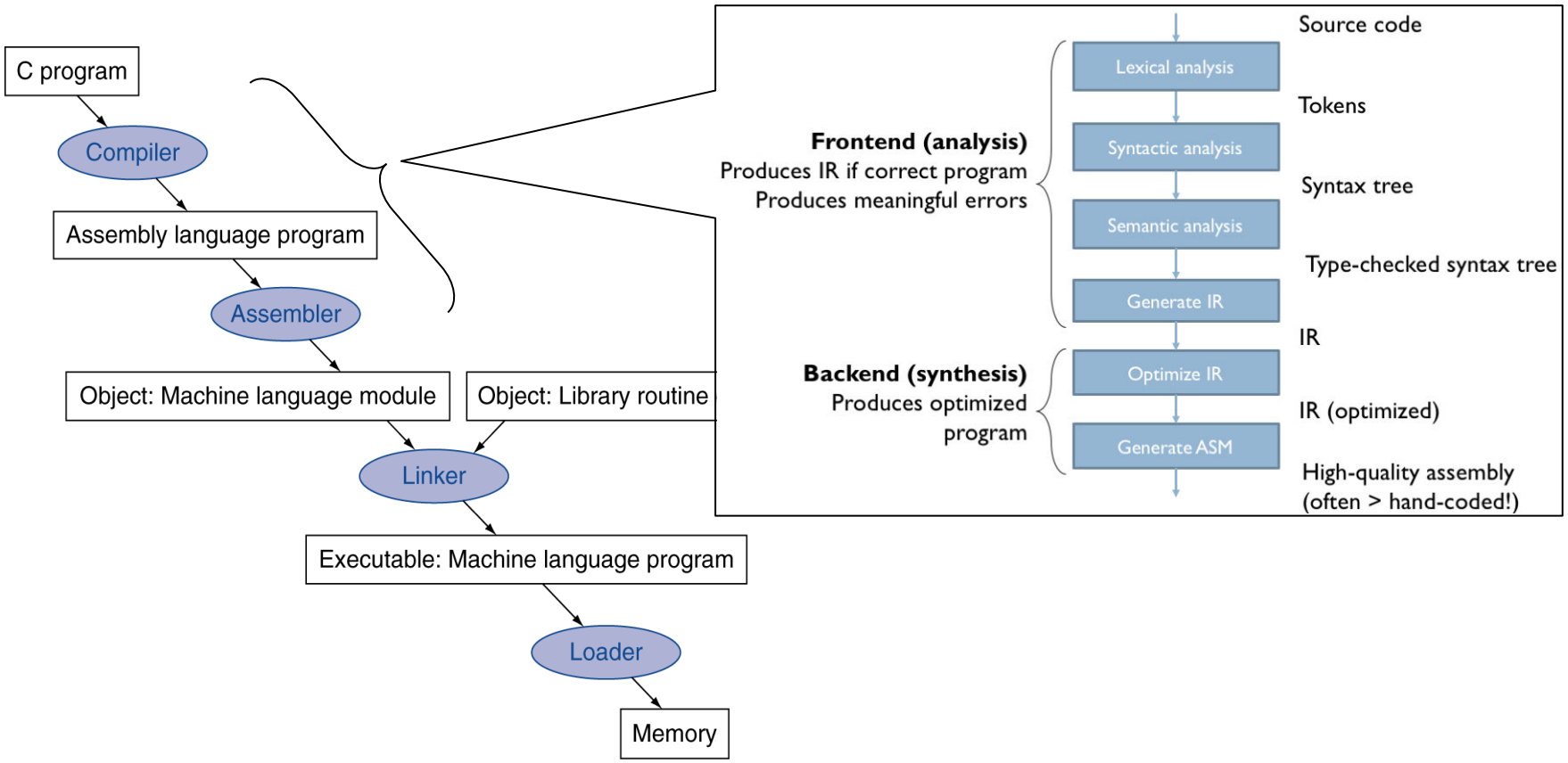
Executable file header		
	Text size	300_{hex}
	Data size	50_{hex}
Text segment	Address	Instruction
	$0040\ 0000_{hex}$	lw $a0, 8000_{hex}($gp)
	$0040\ 0004_{hex}$	jal $40\ 0100_{hex}$
	...	...
	$0040\ 0100_{hex}$	sw $a1, 8020_{hex}($gp)
	$0040\ 0104_{hex}$	jal $40\ 0000_{hex}$
	...	...
Data segment	Address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



Loader

- Load from image file on disk into memory for execution
  1. Read header to determine segment sizes
  2. Create virtual address space
  3. Copy text and initialized data into memory
     - Or set page table entries so they can be faulted in
  4. Set up arguments on stack
  5. Initialize registers (including `$sp, $fp, $gp`)
  6. Jump to startup routine
     - Copies arguments to `$a0`, … and calls main
     - When main returns, do exit `syscall`
- On UNIX systems, loading a program into memory is carried out by the operating system kernel.

Summary



- MIPS implementation of procedure call (stack & heap), array and
- Translating and starting a program
- **Next lecture**: initial ISA implementation (read assignments: Ch3.1 – Ch3.4)