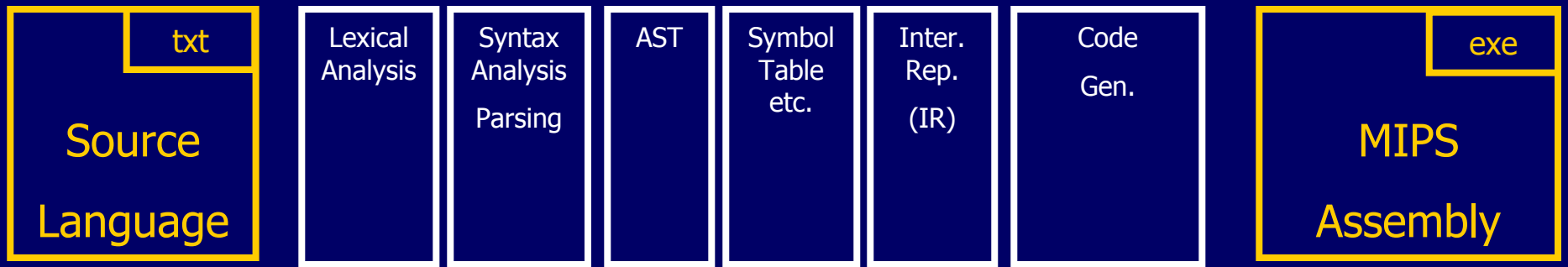
Modern Compiler Design

T10 – MIPS

Mooly Sagiv and Greta Yorsh
School of Computer Science
Tel-Aviv University

gretay@post.tau.ac.il
http://www.cs.tau.ac.il/~gretay

Today



- Goals:
  - MIPS Architecture & Assembly Overview
  - From Source to Executable

MIPS Architecture & Assembly Overview

- Memory
- Registers
- Instructions
- Stack frames
- Examples
  - Hello world
  - Factorial

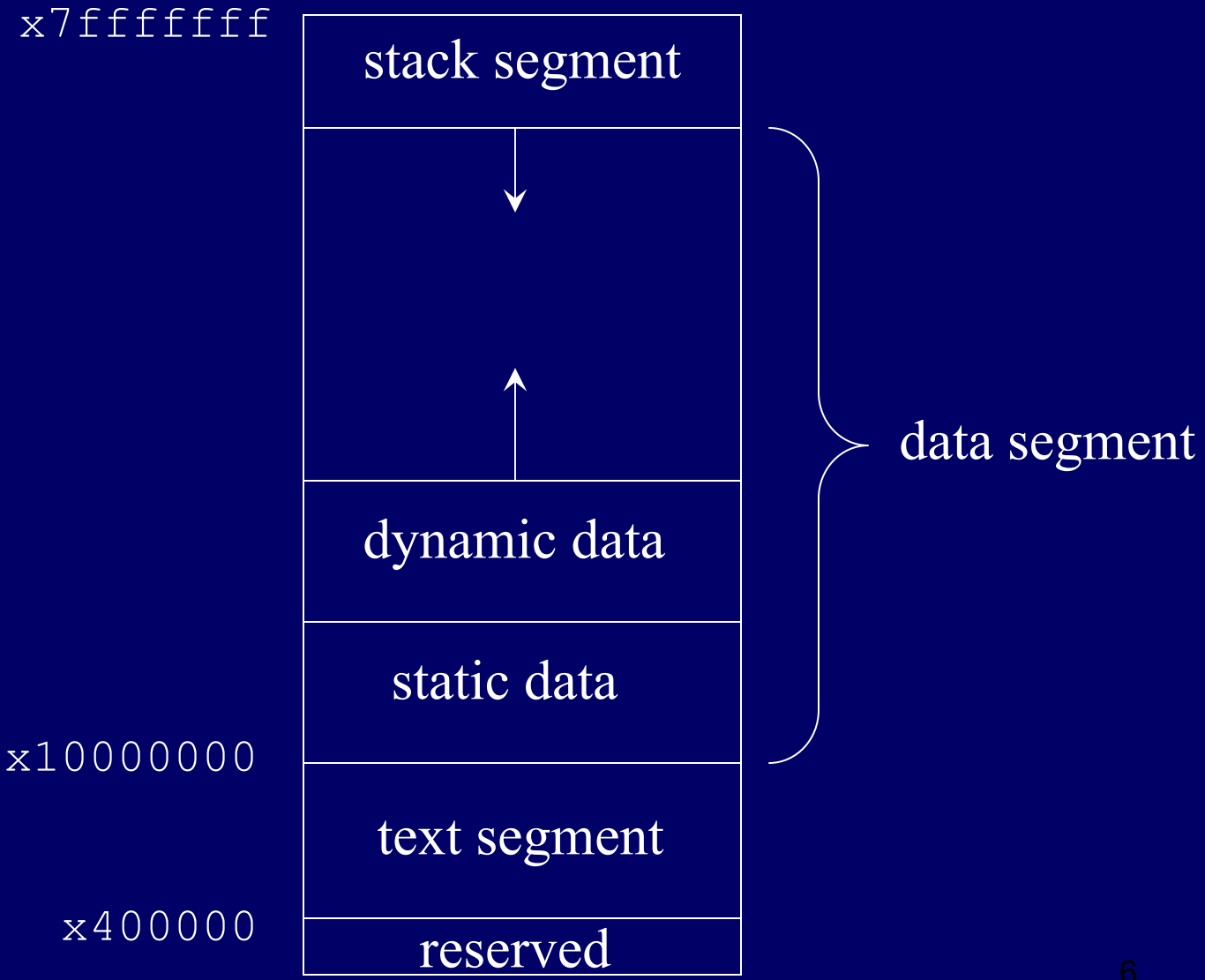
RISC vs. CISC Machines

Feature	RISC	CISC
Registers	∞32	6, 8, 16
Register Classes	One	Some
Arithmetic Operands	Registers	Memory+Registers
Instructions	3-addr	2-addr
Addressing Modes	r M[r+c] (l,s)	several
Instruction Length	32 bits	Variable
Side-effects	None	Some
Instruction-Cost	"Uniform"	Varied

Memory

- Store data and instructions
- Partitioned into bytes, words
- Virtual memory expands physical memory size
- Bounded by machine address size (32bit)

Memory


x7fffffff
stack segment
dynamic data
static data
x10000000
text segment
x400000
reserved
data segment

Gulliver's Travels by Jonathan Swift

- Little Endian
  - Store the little end of the number (least significant bits) first
- Big Endian
  - Store the big end of the number first
- MIPS is little-endian

Example Little Endian

0x1000000007	0x00
0x1000000006	0x00
0x1000000005	0x7f
0x1000000004	0xfe
0x1000000003	0x70('p')
0x1000000002	0x65('l')
0x1000000001	0x65('e')
0x1000000000	0x48('H')

3277 = 0x00007ffe

Help

Machine Registers

- 32 bit register
- General Purpose
  - 32 registers
- Special Purpose
  - a few more

General Purpose Registers

Name	Software Name	Usage
$0		Always 0
$1	at	Reserved for the assembler
$2…$3	v0-v1	Return values
$4…$7	a0-a3	First arguments
$16…$23	s0-s7	Callee-Saved Registers
$24…$25	t8-t9	Caller-Saved Registers
$29	sp	Stack Pointer
$30	fp	Frame Pointer
$31	ra	The return address

Special Purpose Registers

Name	Usage
PC	Program Counter
HI	The most significant 32 bits after multiply and divide
LO	The least significant 32 bits after multiply and divide

Assembly Code

- Allows symbolic names and machine instructions
  - Simplifies the task of the compiler writer
  - Easier to debug the compiler
- One to one translation into machine code
- But may require two passes on the assembly

Instructions: Load and Store

- load from memory to register
- store from register to memory
- load immediate

Instruction	Meaning
li $v0, 4	$v0 ← 4
la $a0, msg	$a0 ← address of msg
lw $t0, x	$t0 ← x
sw $t0, y	y ← $t0

Remarks

- la vs. li
  - Since a label represents a fixed memory address after assembly, la is actually a special case of load immediate.
- lw vs. la
  - x is at address 10 and contains 2

```
la   $a0, x        $a0 ← 10
lw   $a0, x        $a0 ← 2
```

- lw $t0 8($sp)

Instructions: arithmetic and logic operations

- perform the operation on data in 2 registers
- store the result in a 3rd register

Instruction	Meaning
add $t0, $t3, $t4	$t0 ← $t3 + $t4
sub $t0, $t3, $t1	$t0 ← $t3 - $t4
mul $t0, $t3, 35	$t0 ← $t3 * 35

addu, subu, mulu

Instructions: Jump and Branch

Instruction	Meaning
j label	jump to instruction at label
jal my_proc	jump and link start procedure my_proc $ra holds address of instruction following the jal
jr $ra	jump register return from procedure call puts $ra value back into the PC
beq $t0, $t1, label	if ($t0 = $t1) goto label
bneq $t0, $0, label	if ($t0 != $0) goto label
bltz $t0, label	if ($t0 < 0) goto label

bltu, bltuz

Example

```
x = 42;
while (x > 0) {
   x=x-1;
}
```

MISSING CODE !!

(warning: code shown is a naïve code :-)

Instructions

- Operations
  - Load and Store
  - Arithmetic and logical operations
  - Jump and Branch
  - Specialized instructions,
  - Floating-point instructions and registers
- Instruction formats
  - register
  - immediate
  - jump
- Addressing modes
  - immediate - value built in to the instruction
  - register - register used for data
- Memory referencing - used with load and store instructions
  - label - fixed address built in to the instruction
  - indirect - register contains the address
    - Base addressing - field of a record
    - Indexed addressing - element of an array

System Calls

Service	Code	Arguments	Result
print integer	1	$a0=integer	Console print
print string	4	$a0=string address	Console print
read integer	5		$a0=result
read string	8	$a0=string address $a1=length limit	Console read
exit	10		end of program

Hello World

```
        .text           # text segment
        .global __start
__start:                # execution starts here
        la $a0,str      # put string address into a0
        li $v0,4        #
        syscall         # print
        li v0, 10       #
        syscall         # au revoir...
        .data           # data segment
str:    .asciiz  "hello world\n"
```

Riddle

```
        .data
endl:   .asciiz  "\n"

        .text
print_int:
        li   $v0, 1
        syscall
        jr   $ra
endline:
        la   $a0, endl
        li   $v0, 4
        syscall
        jr   $ra
__start:
        li   $a0, 42
        jal print_int
        jal endline
        li   $a0, 2006
        jal print_int
        jal endline
```

Calling Conventions

Stack Frames
Activation Records

Instructions: Jump

Instruction	Meaning
jal my_proc	jump and link start procedure my_proc $ra holds address of instruction following the jal
jr $ra	jump register return from procedure call puts $ra value back into the PC

Parameter Passing

- 1960s
  - In memory
  - No recursion is allowed
- 1970s
  - In stack
- 1980s
  - In registers
  - First k parameters are passed in registers (k=4 or k=6)

Modern Architectures

- Parameters
  - first k parameters are passed in registers, others on the stack
- Return address
  - normally saved in a register on a call (jal)
  - a non-leaf procedure saves this value on the stack
- Function result
  - normally saved in a register on a return
- No stack support in the hardware

Stack Operations in RISC

- PUSH

```
sub   $sp, 4
sw    $ra, ($sp)
```

- POP

```
lw   $ra, ($sp)
add   $sp, 4
```

Stack Frames

- Allocate a separate space for every procedure incarnation
- Provides a simple mean to achieve modularity
- Naturally supports recursion
- LIFO policy
- Efficient memory allocation policy
  - Low overhead

Caller-Save vs. Callee-Save Registers

```
int foo(int a)      {
        int b=a+1;
        f1();
        g1(b);
        return(b+2);
 }
```

```
void bar (int y) {
        int x=y+1;
        f2(y);
        g2(2);
}
```

Calling Convention

- Caller
  - Pass arguments - first 4 are in $a0-$a3 the rest pushed on the stack
  - Save caller-saved registers - including $t0-$t9 if needed
  - jal ($ra gets address of instruction following the jal)
- Callee
  - save callee-saved registers in the frame
    - ☒$fp
    - ☒ $ra (if the callee is not a leaf)
    - ☒ $s0-$s7 (if used by the callee)
  - push a stack frame $fp ← $sp
  - …… do some work
  - restore callee-saved registers
  - pop the stack frame $sp ← $fp
  - jr $ra (puts $ra value back into the PC)

Factorial Example

```
int factorial (int n){
   if (n < 2) return 1;
   return (n * factorial (n-1));  /* n! = n * (n-1)! */
}
factorial:
   bgtz  $a0, doit
   li    $v0, 1        # base case, 0! = 1
   jr    $ra
doit:
   addu $sp, $sp, -8    # stack frame
   sw    $s0,($sp)      # will use for argument n
   sw    $ra,4($sp)     #  return address
   move $s0, $a0        #  save argument
   addu  $a0, $a0, -1           #  n-1
   jal  factorial       #  v0 ← (n-1)!
   mul   $v0,$s0,$v0    #     n*(n-1)!
   lw    $s0,($sp)      # restore registers from stack
   lw    $ra,4($sp)
   addu $sp,$sp,8
   jr    $ra
```

Where is time saved?

- Most procedures are leaf procedures
- Interprocedural register allocation
- Many of the registers may be dead before another invocation
- Register windows are allocated in some architectures per call (e.g., sun Sparc)

Summary

- Understand basic ideas in MIPS architecture
- Can you manually convert a simple C code into a naïve MIPS assembly ?

Code Generation Major Tasks

- Compute the inheritance graph
- Cool Object Layout
  - Assign tags to all classes in depth-first order
  - Determine the layout of attributes, temporaries, and dispatch tables for each class
- Generate code
  - for global data: constants, prototype objects, dispatch tables
  - initialization for each method
  - code for each method