Computer Organization & Design

Hardware/Software interface

楼学庆

Lou Xueqing

玉泉校区曹光彪东楼507室

Website:10.214.47.99/index.php
Email: xqlou@zju.edu.cn
hzlou@163.com

九·一八事变76周年

- 抗日战争的开始
  - 从1931年9月18日，日本关东军完成了向中国军队进攻，向中国百姓动武的最后准备，悍然发动了震惊中外的“九一八”事变。

Homework for Chapter 1&4

- Chapter 1 (p.40)
  - ☞1.46, 1.52, 1.54
- Chapter 4 (p.272)
  - ☞4.2, 4.17, 4.18

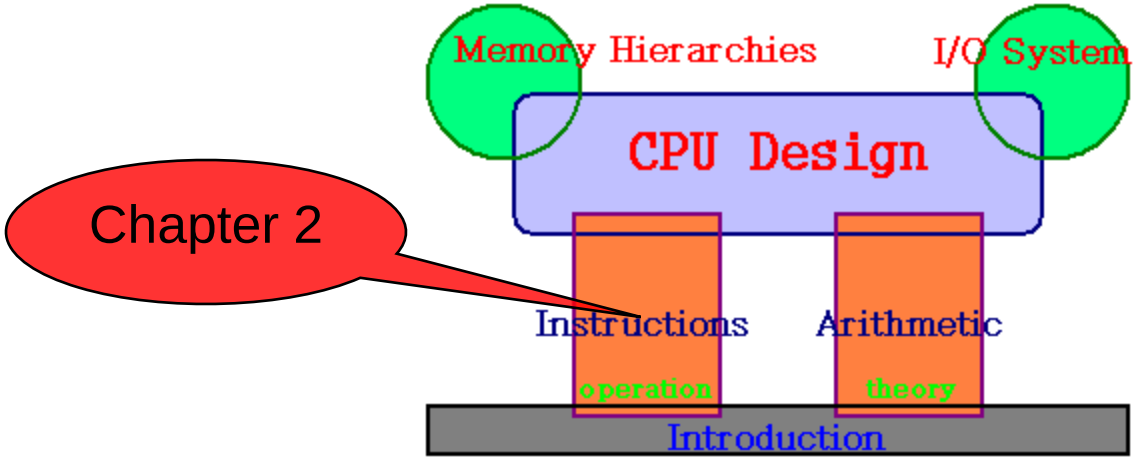
Unit 2:

MIPS Instruction Set

Chapter 2

Chapter 2

- Topics: MIPS instruction set architecture



Introduction

- To command a computer's hardware, you must speak its language.
- The words of a computer's language are called Instructions.
- The vocabulary is called Instruction Set.
- Assembly language:

  a symbolic representation of machine instructions.
- Machine language:

  the numeric instructions.

- The process of compiling and assembling

High-level language program (in C)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

C compiler

Assembly language program (for MIPS)

```
swap:
    muli $2, $5,4
    add  $2, $4,$2
    lw   $15, 0($2)
    lw   $16, 4($2)
    sw   $16, 0($2)
    sw   $15, 4($2)
    jr   $31
```

a symbolic representation of machine instructions

Assembler

Binary machine language program (for MIPS)

```
00000000101000010000000000011000
00000000100011100001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

von Neumann' Computer

- Today's computers are built on 2 key principles:
  - ☞①Instruction are represented as numbers.
  - ☞②Programs can be stored in memory to be read or written just like numbers.

- **http://10.214.47.99:8080/mips/**

Four Design Principles

- 1. Simplicity favors regularity
- 2. Smaller is faster
- 3. Good design demands good compromises
- 4. Make the common case fast

Recap

- Knowledge of hardware improves software quality: compilers, OS, threaded programs, memory management
- Important trends: growing transistors, move to multi-core, slowing rate of performance improvement, power/thermal constraints, long memory/disk latencies

Instruction Set

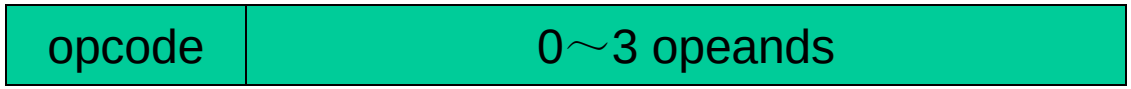
- Understanding the language of the hardware is key to understanding the hardware/software interface
- A program (in say, C) is compiled into an executable that is composed of machine instructions – this executable must also run on future machines – for example, each Intel processor reads in the same x86 instructions, but each processor handles instructions differently
- Java programs are converted into portable bytecode that is converted into machine instructions during execution (just-in-time compilation)
- What are important design principles when defining the instruction set architecture (ISA)?

Instruction Set

- Important design principles when defining the instruction set architecture (ISA):
  - keep the hardware simple – the chip must only implement basic primitives and run fast
  - keep the instructions regular – simplifies the decoding/scheduling of instructions

Assembly code

- Opcode: the field that denotes the operation and format of an instruction.
  - ☞add, sub, lw, sw, beq, …
- Operand
  - ☞register, memory, immediate, …
- Assembly instruction



- ☞Add x, y, z
  - ✧x = y + z
- ☞Sub a, b, c
  - ✧a = b - c

The natural number of operands for an operation like addition is 3.

Requiring every instruction to have exactly 3 operands, no more and no less.

A Basic MIPS Instruction

C code: a = b + c ;

The translation from C to MIPS assembly language instructions is performed by the ***compiler***.

Assembly code: (human-friendly machine instructions, a symbolic representation of machine instructions)

add a, b, c # a is the sum of b and c

The translation from MIPS to machine code is performed by the ***assembler***.

Machine code: (hardware-friendly machine instructions)

00000010001100100000000100000

Translate the following C code into assembly code:

a = b + c + d + e;

- Translate the following C code into assembly code:

  a = b + c + d + e;

- The operands of arithmetic instructions are restricted.

Example

C code a = b + c + d + e;

translates into the following assembly code:

```
add  a, b, c          add  a, b, c
add  a, a, d    or    add  f, d, e
add  a, a, e          add  a, a, f
```

- Instructions are simple: fixed number of operands (unlike C)
- A single line of C code is converted into multiple lines of assembly code
- Some sequences are better than others… the second sequence needs one more (temporary) variable f

Subtract Example

C code f = (g + h) – (i + j);

Assembly code translation with only add and sub instructions:

Subtract Example

C code f = (g + h) – (i + j);
translates into the following assembly code:

```
add  t0, g, h          add  f, g, h
add  t1,  i, j    or   sub   f, f, i
sub  f,   t0, t1       sub   f, f, j
```

- Each version may produce a different result because floating-point operations are not necessarily associative and commutative… more on this later

Operands

- In C, each “variable” is a location in memory

- In hardware, each memory access is expensive – if variable *a* is accessed repeatedly, it helps to bring the variable into an on-chip scratchpad and operate on the scratchpad (registers)

- To simplify the instructions, we require that each instruction (add, sub) only operate on registers

- Note: the number of operands (variables) in a C program is very large; the number of operands in assembly is fixed…
  there can be only so many scratchpad registers

Registers

- The MIPS ISA has 32 registers (x86 has 8 registers) – Why not more? Why not less?
- Each register is 32-bit wide (modern 64-bit architectures have 64-bit wide registers)
- A 32-bit entity (4 bytes) is referred to as a word
- To make the code more readable, registers are partitioned as $s0-$s7 (C/Java variables), $t0-$t9 (temporary variables)…

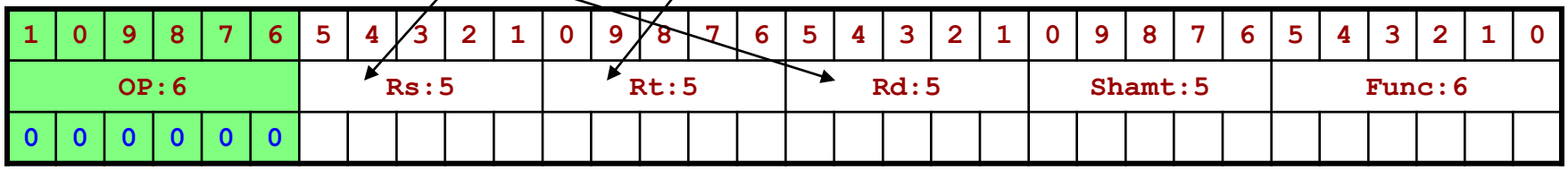
Policy of Use Conventions

Name	Register number	Usage
$zero	0	the constant value 0
$at	1	Reserve for assmbler
$v0-$v1	2-3	values for results and expression evaluation
$a0-$a3	4-7	arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved
$t8-$t9	24-25	more temporaries
$k0-$k1	26-27	Reserve for Operating
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

R-type Instruction

- R-Type - This group contains all instructions that do not require an immediate value, target offset, memory address displacement, or memory address to specify an operand. This includes arithmetic and logic with all operands in registers, shift instructions, and register direct jump instructions (jalr and jr).
- All R-type instructions use opcode **000000**.

add $t0, $s1, $s2



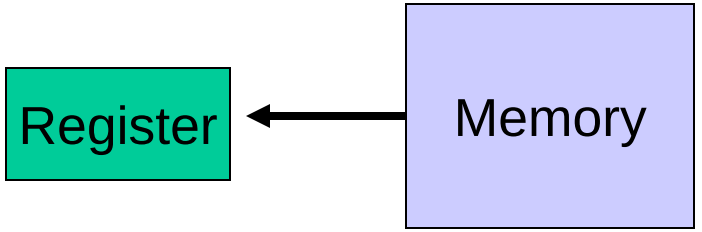
Registers

- The MIPS CPU contains 32 general-purpose registers that are numbered 0–31.
  - $0：always contains the hardwired value 0.
  - Registers $at (1), $k0 (26), and $k1 (27) are reserved for the assembler and operating system and should not be used by user programs or compilers.
  - Registers $a0–$a3 (4–7) are used to pass the first four arguments to routines (remaining arguments are passed on the stack).
  - Registers $v0 and $v1(2, 3) are used to return values from functions.

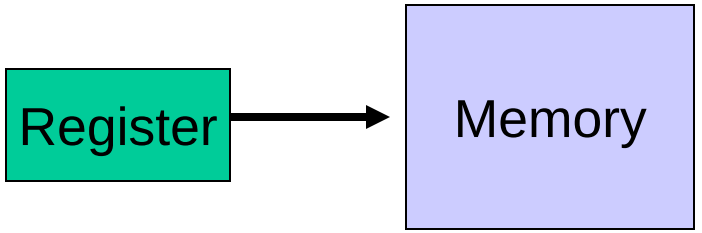
Memory Operands

- Values must be fetched from memory before (add and sub) instructions can operate on them

Load word
lw $t0, memory-address



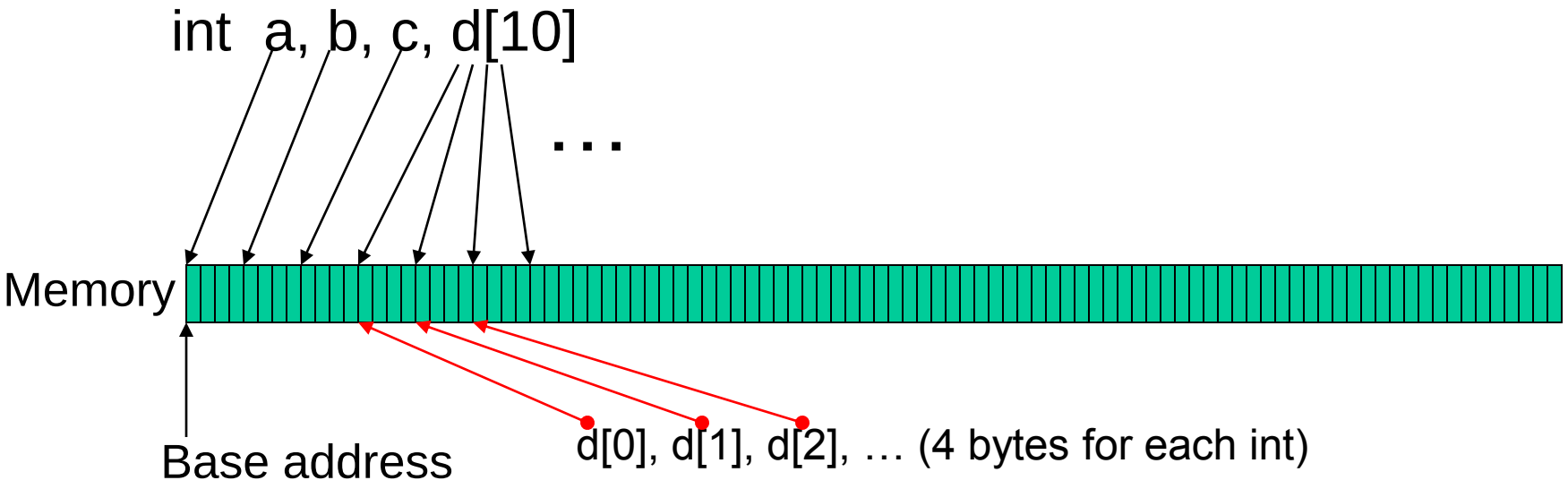
Store word
sw $t0, memory-address



How is memory-address determined?

Memory Address

- The compiler organizes data in memory… it knows the location of every variable (saved in a table)… it can fill in the appropriate mem-address for load-store instructions(L/S)



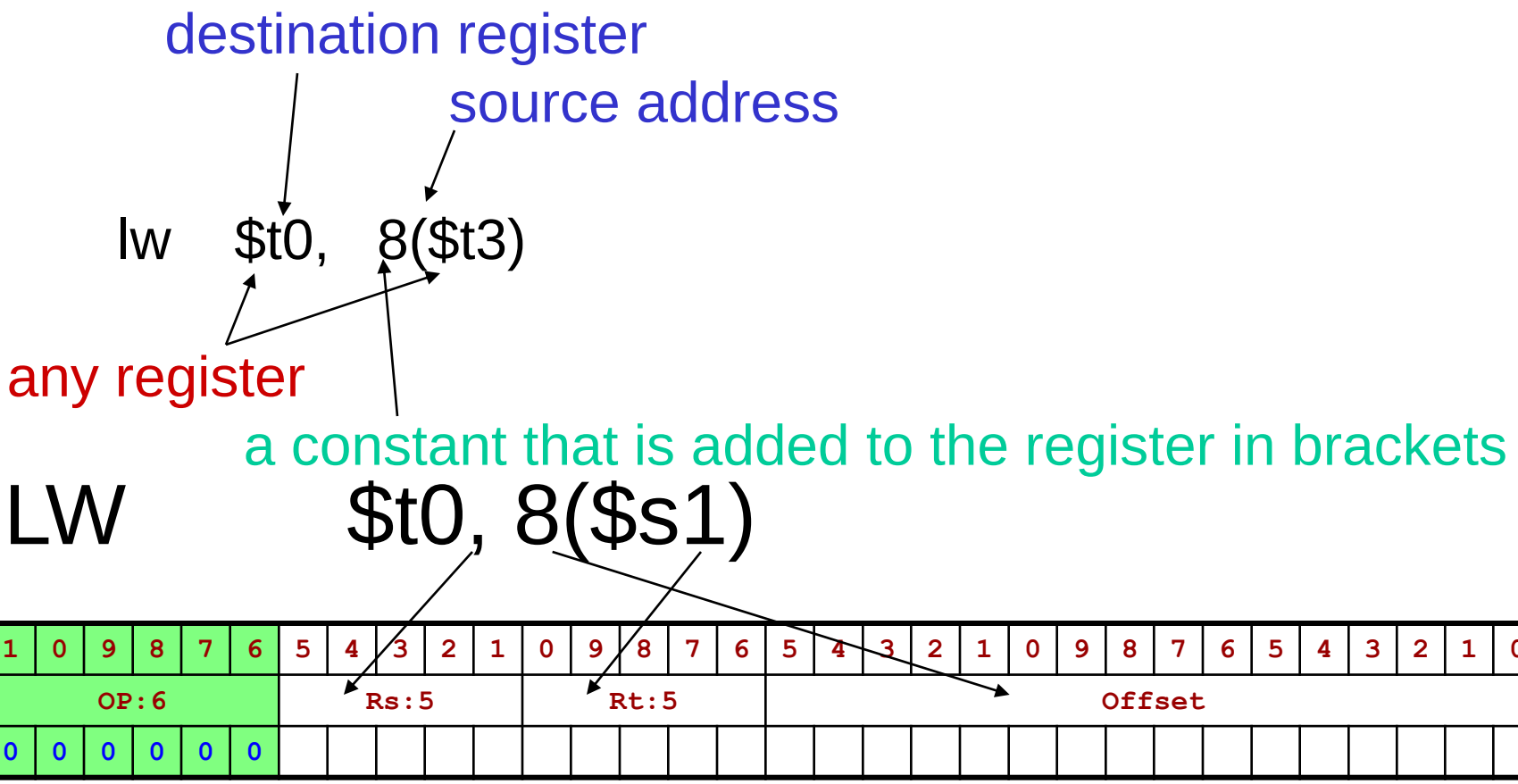
Immediate Operands

- An instruction may require a constant as input
- An immediate instruction uses a constant number as one of the inputs (instead of a register operand)

```
addi   $s0, $zero, 1000   # the program has base address
                          #  1000 and this is saved in $s0
                          # $zero is a register that always
                          # equals zero
addi   $s1, $s0, 0        # this is the address of variable a
addi   $s2, $s0, 4        # this is the address of variable b
addi   $s3, $s0, 8        # this is the address of variable c
addi   $s4, $s0, 12       # this is the address of variable d[0]
```

I-type Instruction

- The format of a load instruction:



Example

Convert to assembly:

C code: d[3] = d[2] + a;

Example

Convert to assembly:

C code: d[3] = d[2] + a;

Assembly: # addi instructions as before

```
lw   $t0, 8($s4)   #  d[2] is brought into $t0
lw   $t1, 0($s1)   #   a  is brought into $t1
add  $t0, $t0, $t1  #  the sum is in $t0
sw   $t0, 12($s4)  #  $t0 is stored into d[3]
```

Assembly version of the code continues to expand!

Recap – Numeric Representations

- Decimal 35_{10}
- Binary 00100011_2
- Hexadecimal (compact representation)

  0x 23 or 23_{hex}

  0-15 (decimal) → 0-9, a-f (hex)

Instruction Formats

Instructions are represented as 32-bit numbers (one word), broken into 6 fields

R-type instruction add $t0, $s1, $s2

000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
op	rs	rt	rd	shamt	funct
opcode	source	source	dest	shift amt	function

I-type instruction lw $t0, 32($s3)

6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	constant

Logical Operations

Logical ops	C operators	Java operators	MIPS instr
Shift Left	<<	<<	sll
Shift Right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit NOT	~	~	nor

Instructions

- R-type
  - ☞Add, Sub, Or, And, …
- I-type
  - ☞LW, SW, BEQ, BNE, ADDi, …
- J-type
  - ☞J, …

Exercise(Sep.25)

- Assemble
  - `ADD $S2, $T8, $T0`
  - `LW $S0, $S1(-123)`
  - `SW $RA, $SP(123)`
  - `For: BEQ $T0, $T1, For`
- Un-assemble
  - `02488824`
  - `8E30FF85`
  - `AFBF007B`
  - `1109FFFF`

Making Decision

- Based on the input data and the value created during computation, different instructions execute.
- Unconditional branch
  - J
  - JR, JAL
- Conditional branches
  - BEQ, BNE
  - SLT
  - …

Control Instructions

- Conditional branch: Jump to instruction L1 if register1 equals register2: beq register1, register2, L1
  Similarly, bne and slt (set-on-less-than)

- Unconditional branch:
  j L1
  jr $s0

Convert to assembly:

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

```
        beq    I, j, Equal
        sub    f, g, h
        j      Exit
Equal: add     f, g, h
Exit:
```

Control Instructions

- Conditional branch: Jump to instruction L1 if register1 equals register2: beq register1, register2, L1
  Similarly, bne and slt (set-on-less-than)

- Unconditional branch:

```
j       L1
jal     label
jr      $s0
```

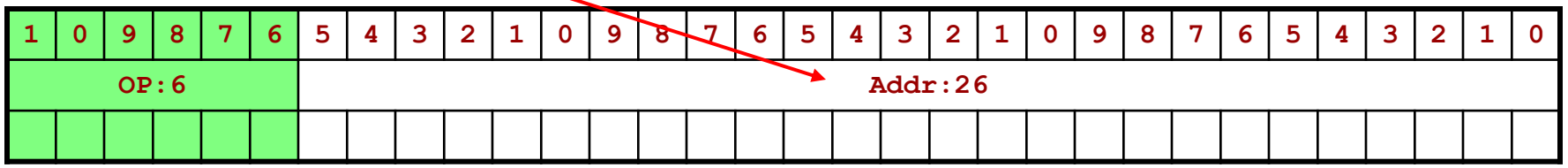
Convert to assembly:

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

```
        beq    I, j, Equal
        sub    f, g, h
        j      Exit
Equal: add     f, g, h
Exit:
```

J: Jump instruction

- J label



- Execute:
  - ☞PC = label
    - ✧Direct addressing. but ***impossible***, why?
  - ☞PC = ((PC+4) & 0xF000_0000) | (label << 2)
    - ✧Pseudodirect addressing
  - ☞PC: Program Count
    - ✧The register that always holds the address of the current instruction being executed.

J-type

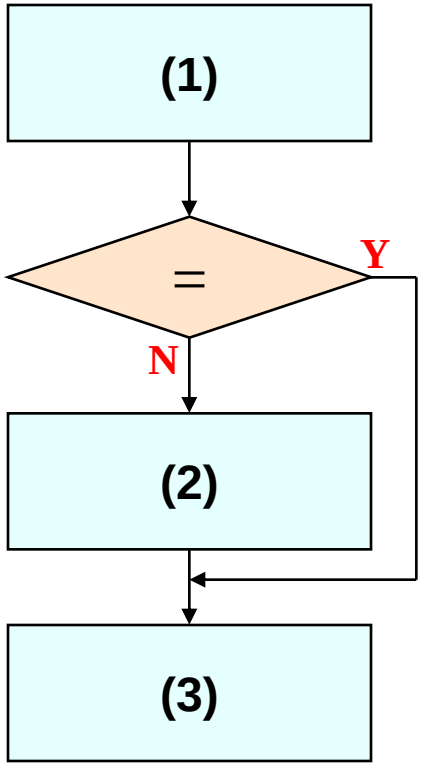
- J-Type - This group consists of the two direct jump instructions (j and jal). These instructions require a memory address to specify their operand.
- J-type instructions use opcodes 00001x.
- JAL: Jump and Link
  - ☞Execute:
    - ✧1. $ra = PC+4;
    - ✧2. J label;
- JR $r
  - ☞Execute:
    - ✧PC = $r

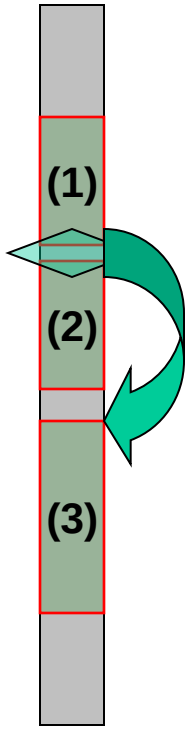
Control Instructions

- J LABEL

```
        beq    I, j, Equal
        sub    f, g, h
        j      Exit
Equal:  add    f, g, h
Exit:
```

Conditional branch



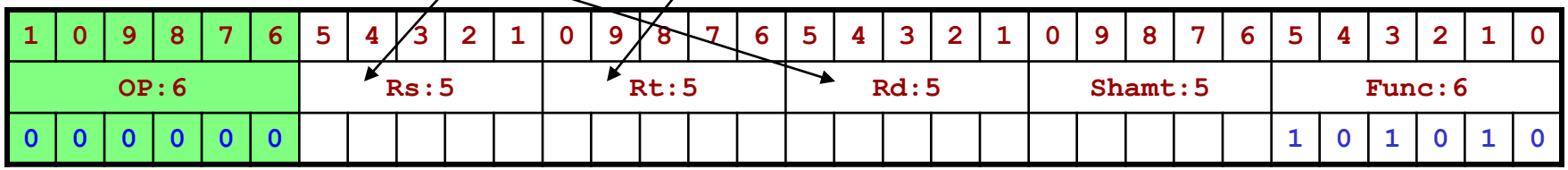



```
(1)    bne   $s3, $s4, Exit
(2)    add   $s0, $s1, $s2
Exit: (3)
```

SLT

- SLT $rd, $r1, $r2
  - ☞if($r1 < $r2) $rd = 1; else $rd = 0;

slt $t0, $s1, $s2



BEQ & BNE

- Beq $r1, $r2, Label
  - ☞if($r1 == $r2) goto Label
    - ✧goto Label: PC=PC+4+(Label<<2)

```
-- PC relative addressing
-- $r: any register
```

- Bne $r1, $r2, Label
  - ☞if($r1 != $r2) goto Label

<、 >、 <=、 >=

- if($s0 < $s1) goto Label

```
Slt   $t0, $s0, $s1
Bne   $t0, $zero, Label
```

- if($r0 > $r1) goto Label

```
Slt   $at, $r1, $r0
Bne   $at, $zero, Label
```

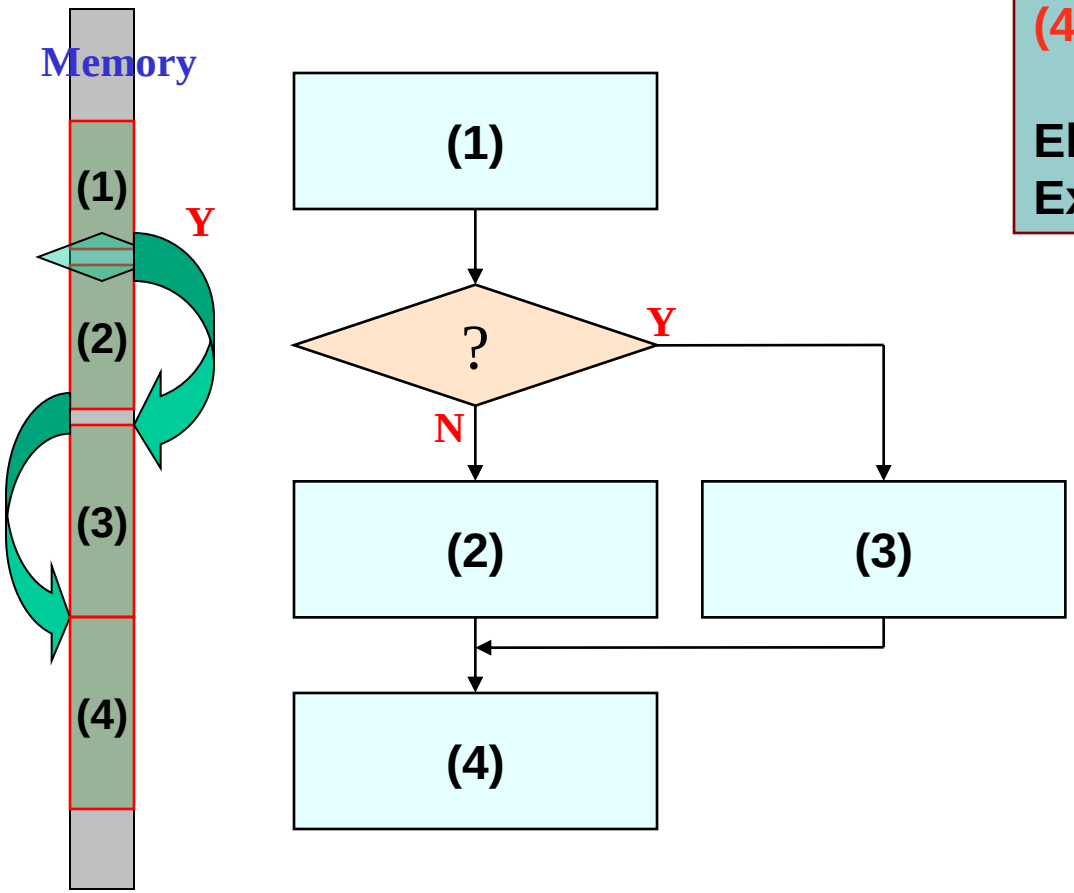
- if($r0 >= $r1) goto Label

```
Slt   $at, $r1, $r0
Beq   $at, $zero, Label
```

- if($r0 <= $r1) goto Label

```
Slt   $at, $r1, $r0
Beq   $at, $zero, Label
```

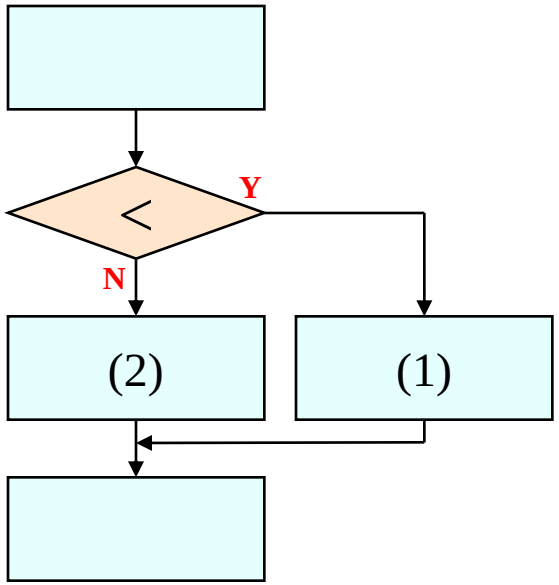
Conditional branch




```
(1)     bne   $s3, $s4, Else
(4)     add   $s0, $s1, $s2
        j     Exit
Else:   (2)sub  $s0, $s1, $s2
Exit:   (3)
```

Control Flow

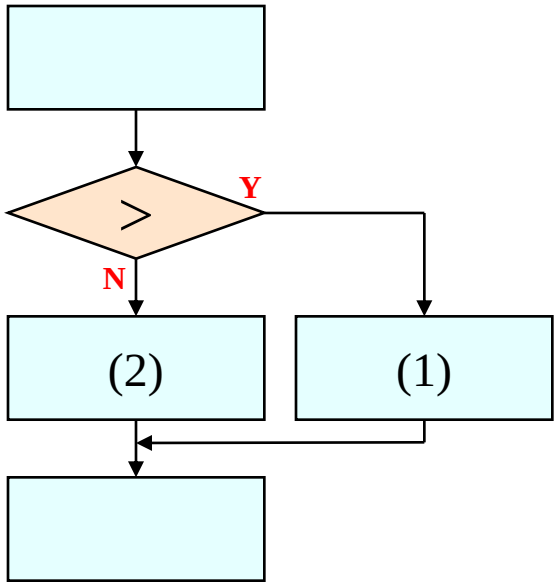
- if ($s1 < $s2) then
  …(1)
  else
  …(2)




```
        Slt   $t0, $s1, $s2
        bne   $t0, $zero, (1)
              …(2)
        j     exit
(1)           …(1)
Exit:
        ……
```

Control Flow

- if ($s1 > $s2) then
  …(1)
  else
  …(2)




```
        Slt   $t0, $s2, $s1
        bne   $t0, $zero, (1)
              …(2)
        j     exit
(1)           …(1)
Exit:
        ……
```

SLT

Pseudo instruction

- SLT $rd, $r1, $r2
  - if($rs<$rt)$rd=1; else $rd=0;
- if($r1 < $r2)goto lable
  - Blt $r1, $r2, label

```
SLT $at, $r1, $r2
Bne $at, $zero, label
```

- if($r1 > $r2)goto lable
  - Bgt $r1, $r2, label

```
SLT $at, $r2, $r1
Bne $at, $zero, label
```

- if($r1<=$r2)goto lable
  - Ble $r1, $r2, label

```
SLT $at, $r2, $r1
Beq $at, $zero, label
```

- if($r1>=$r2)goto lable
  - Bge $r1, $r2, label

```
SLT $at, $r1, $r2
Beq $at, $zero, label
```

Pseudo instruction

- These instructions need not be implemented in hardware; however, their appearance in assembly language simplifies translation and programming.
  - ☞When considering performance you should count real instructions.

Add $r1, $r2, $zero

- e.g. ($r, $r1, $r2: any register)
  - ☞Move $r1, $r2 //$r1=$r2

Beq $r, $zero, label

  - ☞Beqz $r, label

Pseudo instruction

- These instructions need not be implemented in hardware; however, their appearance in assembly language simplifies translation and programming.
  - ☞When considering performance you should count real instructions.
- e.g. ($r, $r1, $r2: any register)
  - ☞Move $r1, $r2 //$r1=$r2

    Add $r1, $r2, $zero
  - ☞Beqz $r, label

    Beq $r, $zero, label
  - ☞……

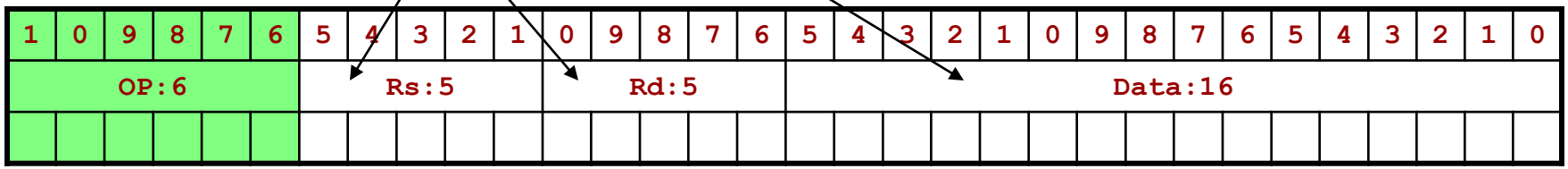
I-Type

- This group includes instructions with an immediate operand
  - ☞branch instructions
  - ☞load and store instructions
- In the MIPS architecture, all memory accesses are handled by the main processor, so coprocessor load and store instructions are included in this group.
- All opcodes except 000000, 00001x, and 0100xx are used for I-type instructions.

I-type Instruction

- I: immediate

```
beq    $t0, $s1, exit
lw     $t0, 0($s1)
sw     $t0, 0($s1)
```



Control Instructions

- Conditional branch: Jump to instruction L1 if register1 equals register2: beq register1, register2, L1
  Similarly, bne and slt (set-on-less-than)

- Unconditional branch:
  j L1
  jr $s0

Convert to assembly:

```
if (i == j)                    bne  $s3, $s4, Else
    f = g+h;                   add  $s0, $s1, $s2
else                           j    Exit
    f = g-h;              Else: sub  $s0, $s1, $s2
                          Exit:
```

Example

Convert to assembly:

```
while   (save[i] == k)
    i += 1;
```

i and k are in $s3 and $s5 and
base of array save[] is in $s6

Example

Convert to assembly:

```
while   (sa[i] == k)
    i += 1;
```

i and k are in $s3 and $s5 and base of array sa[] is in $s6

```
Loop:  sll    $t1, $s3, 2
       add    $t1, $t1, $s6
       lw     $t0, 0($t1)
       bne    $t0, $s5, Exit
       addi   $s3, $s3, 1
       j      Loop
Exit:
```

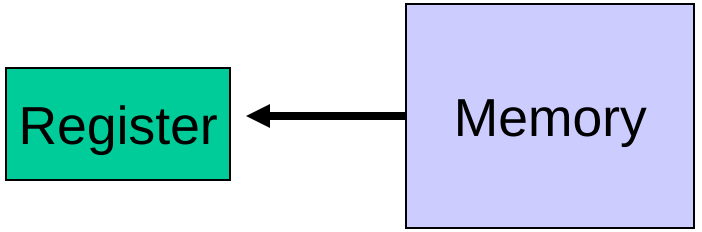
Lecture: MIPS Instruction Set

- The topic:
  - More MIPS instructions
  - Procedure call/return
- Reminder:

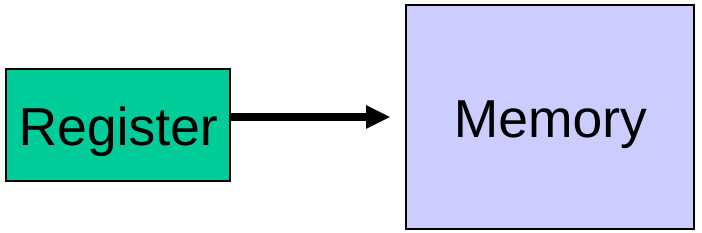
Memory Operands

- Values must be fetched from memory before (add and sub) instructions can operate on them

Load word
lw $t0, memory-address



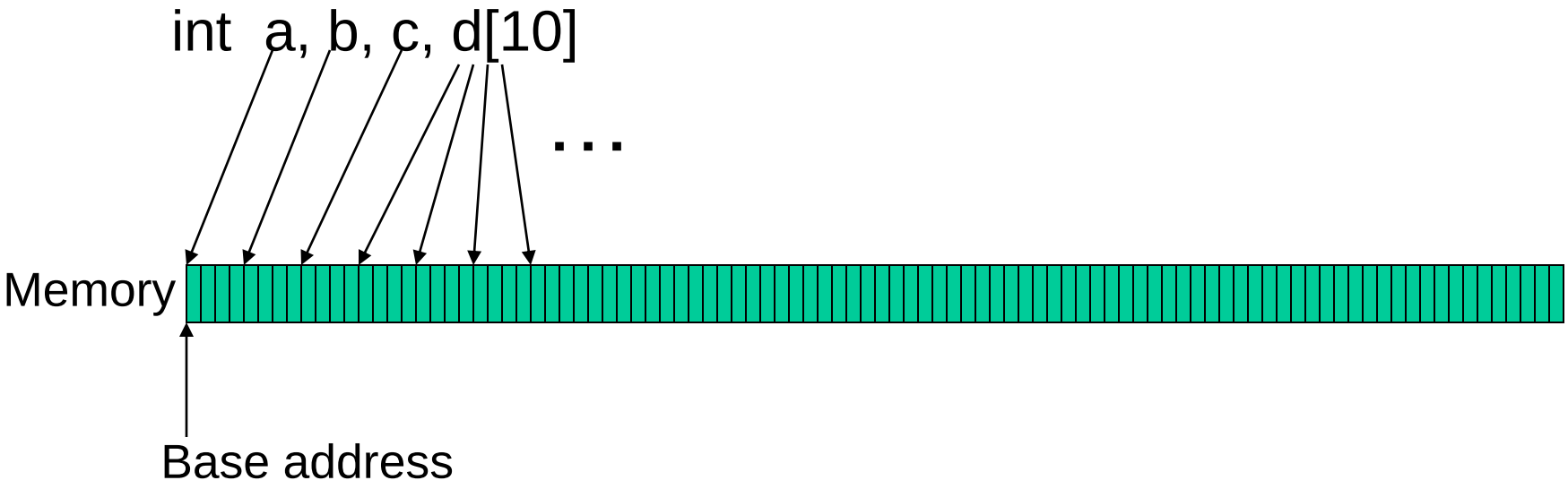
Store word
sw $t0, memory-address



How is memory-address determined?

Memory Address

- The compiler organizes data in memory… it knows the location of every variable (saved in a table)… it can fill in the appropriate mem-address for load-store instructions



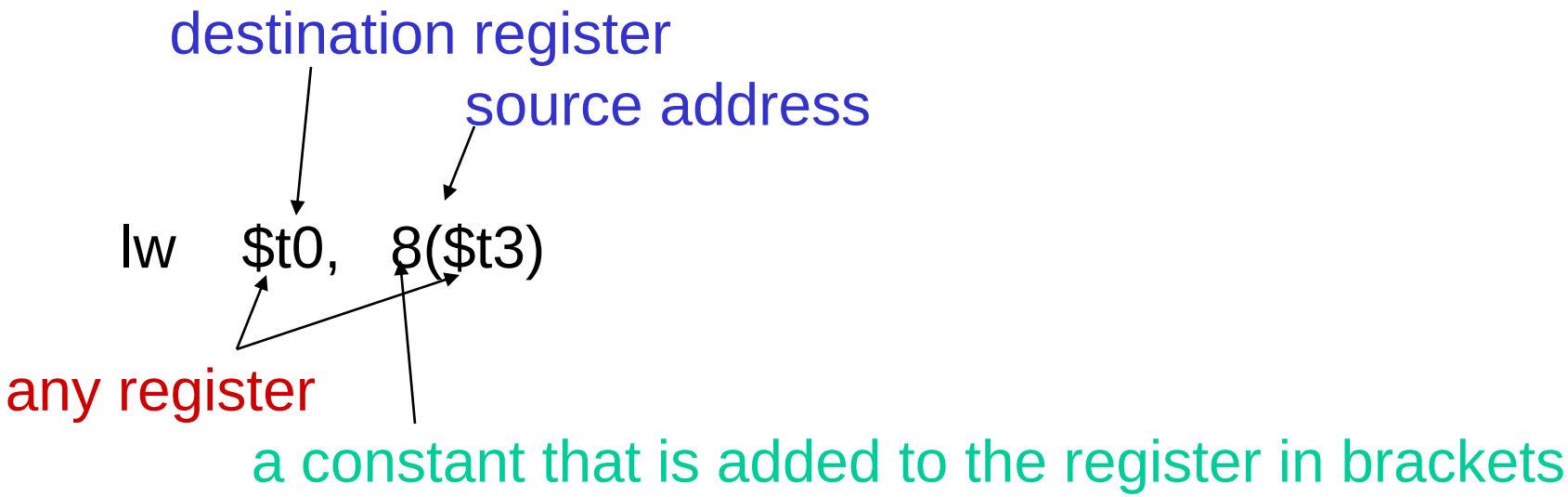
Immediate Operands

- An instruction may require a constant as input
- An immediate instruction uses a constant number as one of the inputs (instead of a register operand)

```
addi   $s0, $zero, 1000   # the program has base address
                          #  1000 and this is saved in $s0
                          # $zero is a register that always
                          # equals zero
addi   $s1, $s0, 0        # this is the address of variable a
addi   $s2, $s0, 4        # this is the address of variable b
addi   $s3, $s0, 8        # this is the address of variable c
addi   $s4, $s0, 12       # this is the address of variable d[0]
```

Memory Instruction Format

- The format of a load instruction:



Example

Convert to assembly:

C code: d[3] = d[2] + a;

Assembly: # addi instructions as before

```
lw   $t0, 8($s4)   #  d[2] is brought into $t0
lw   $t1, 0($s1)   #   a  is brought into $t1
add  $t0, $t0, $t1  #  the sum is in $t0
sw   $t0, 12($s4)  #  $t0 is stored into d[3]
```

Assembly version of the code continues to expand!

Recap – Numeric Representations

- Decimal $35_{10} = 3 \times 10^1 + 5 \times 10^0$
- Binary $00100011_2 = 1 \times 2^5 + 1 \times 2^1 + 1 \times 2^0$
- Hexadecimal (compact representation)

  $0x\ 23$ or $23_{hex} = 2 \times 16^1 + 3 \times 16^0$

  0-15 (decimal) → 0-9, a-f (hex)

Dec	Binary	Hex	Dec	Binary	Hex	Dec	Binary	Hex	Dec	Binary	Hex
0	0000	00	4	0100	04	8	1000	08	12	1100	0c
1	0001	01	5	0101	05	9	1001	09	13	1101	0d
2	0010	02	6	0110	06	10	1010	0a	14	1110	0e
3	0011	03	7	0111	07	11	1011	0b	15	1111	0f

Instruction Formats

Instructions are represented as 32-bit numbers (one word), broken into 6 fields

R-type instruction add $t0, $s1, $s2

000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
op	rs	rt	rd	shamt	funct
opcode	source	source	dest	shift amt	function

I-type instruction lw $t0, 32($s3)

6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	constant

Logical Operations

Logical ops	C operators	Java operators	MIPS instr
Shift Left	<<	<<	sll
Shift Right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit NOT	~	~	nor

Control Instructions

- Conditional branch: Jump to instruction L1 if register1 equals register2: beq register1, register2, L1
  Similarly, bne and slt (set-on-less-than)

- Unconditional branch:
  j L1
  jr $s0 (useful for large case statements and big jumps)

Convert to assembly:

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

Control Instructions

- Conditional branch: Jump to instruction L1 if register1 equals register2: beq register1, register2, L1
  Similarly, bne and slt (set-on-less-than)

- Unconditional branch:
  j L1
  jr $s0 (useful for large case statements and big jumps)

Convert to assembly:

```
if (i == j)                    bne  $s3, $s4, Else
    f = g+h;                   add  $s0, $s1, $s2
else                           j    Exit
    f = g-h;              Else: sub  $s0, $s1, $s2
                          Exit:
```

Example

Convert to assembly:

```
while   (save[i] == k)
    i += 1;
```

i and k are in $s3 and $s5 and
base of array save[] is in $s6

Example

Convert to assembly:

```
while   (save[i] == k)
     i += 1;
```

i and k are in $s3 and $s5 and base of array save[] is in $s6

```
Loop:  sll    $t1, $s3, 2
       add    $t1, $t1, $s6
       lw     $t0, 0($t1)
       bne    $t0, $s5, Exit
       addi   $s3, $s3, 1
       j      Loop
Exit:
```

Procedures

- Each procedure (function, subroutine) maintains a scratchpad of register values – when another procedure is called (the callee), the new procedure takes over the scratchpad – values may have to be saved so we can safely return to the caller
  - parameters (arguments) are placed where the callee can see them
  - control is transferred to the callee
  - acquire storage resources for callee
  - execute the procedure
  - place result value where caller can access it
  - return control to caller

Registers

- The 32 MIPS registers are partitioned as follows:
  - Register 0 : $zero always stores the constant 0
  - Regs 2-3 : $v0, $v1 return values of a procedure
  - Regs 4-7 : $a0-$a3 input arguments to a procedure
  - Regs 8-15 : $t0-$t7 temporaries
  - Regs 16-23: $s0-$s7 variables
  - Regs 24-25: $t8-$t9 more temporaries
  - Reg 28 : $gp global pointer
  - Reg 29 : $sp stack pointer
  - Reg 30 : $fp frame pointer
  - Reg 31 : $ra return address

Jump-and-Link

- A special register (storage not part of the register file) maintains the address of the instruction currently being executed – this is the *program counter* (PC)

- The procedure call is executed by invoking the jump-and-link (jal) instruction – the current PC (actually, PC+4) is saved in the register $ra and we jump to the procedure's address (the PC is accordingly set to this address)

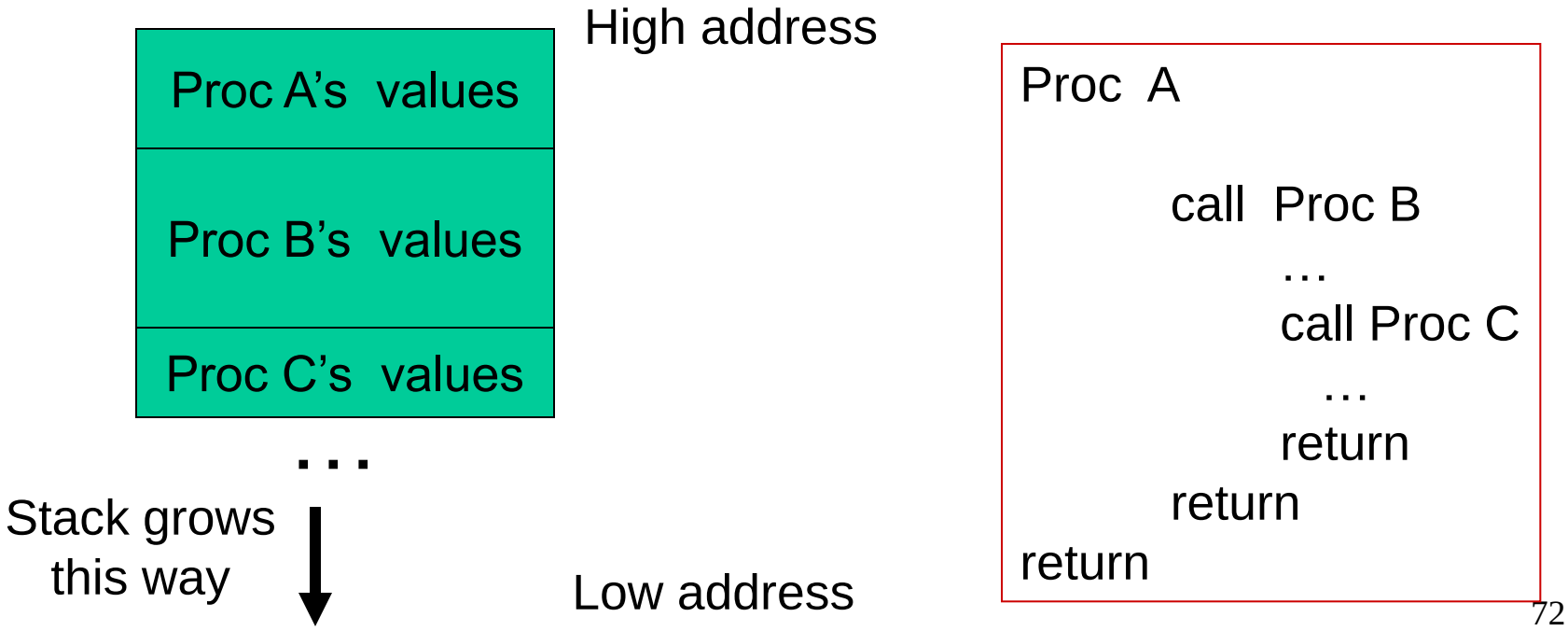
jal NewProcedureAddress

- Since jal may over-write a relevant value in $ra, it must be saved somewhere (in memory?) before invoking the jal instruction

- How do we return control back to the caller after completing the callee procedure?

The Stack

The register scratchpad for a procedure seems volatile – it seems to disappear every time we switch procedures – a procedure's values are therefore backed up in memory on a stack



Storage Management on a Call/Return

- A new procedure must create space for all its variables on the stack
- Before executing the jal, the caller must save relevant values in $s0-$s7, $a0-$a3, $ra, temps into its own stack space
- Arguments are copied into $a0-$a3; the jal is executed
- After the callee creates stack space, it updates the value of $sp
- Once the callee finishes, it copies the return value into $v0, frees up stack space, and $sp is incremented
- On return, the caller may bring in its stack values, ra, temps into registers
- The responsibility for copies between stack and registers may fall upon either the caller or the callee

Saves on Stack

- Caller saved
  - $a0-a3 -- old arguments must be saved before setting new arguments for the callee
  - $ra -- must be saved before the jal instruction over-writes this value
  - $t0-t9 -- if you plan to use your temps after the return, save them note that callees are free to use temps as they please
  - You need not save $s0-s7 as the callee will take care of them

- Callee saved
  - $s0-s7 -- before the callee uses such a register, it must save the old contents since the caller will usually need it on return
  - local variables -- space is also created on the stack for variables local to that procedure

Example 1

```
int  leaf_example (int g, int h, int i, int j)
{
    int f ;
    f = (g + h) – (i + j);
    return f;
}
```

Example 1

```
int  leaf_example (int g, int h, int i, int j)
{
    int f ;
    f = (g + h) – (i + j);
    return f;
}
```

```
leaf_example:
   addi    $sp,  $sp,  -12
   sw      $t1, 8($sp)
   sw      $t0, 4($sp)
   sw      $s0, 0($sp)
   add     $t0, $a0, $a1
   add     $t1, $a2, $a3
   sub     $s0, $t0, $t1
   add     $v0, $s0, $zero
   lw      $s0, 0($sp)
   lw      $t0, 4($sp)
   lw      $t1, 8($sp)
   addi    $sp, $sp, 12
   jr      $ra
```

Notes:
In this example, the procedure's stack space was used for the caller's variables, not the callee's – the compiler decided that was better.

The caller took care of saving its $ra and $a0-$a3.

Example 2

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

Example 2

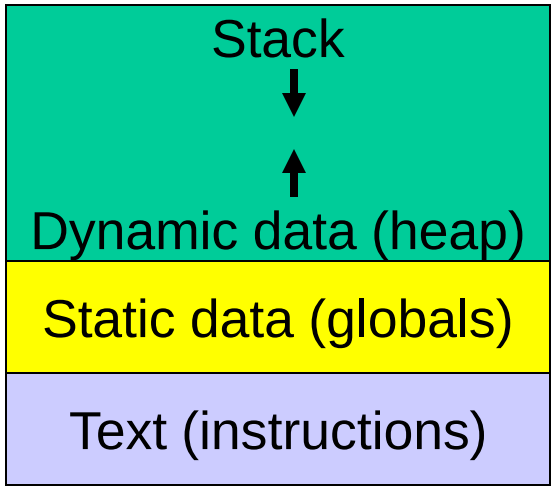
```
int   fact  (int n)
{
     if (n < 1)  return (1);
          else return (n * fact(n-1));
}
```

Notes:
The caller saves $a0 and $ra in its stack space.
Temps are never saved.

```
fact:
    addi     $sp, $sp, -8
    sw       $ra, 4($sp)
    sw       $a0, 0($sp)
    slti     $t0, $a0, 1
    beq      $t0, $zero, L1
      addi   $v0, $zero, 1
      addi   $sp, $sp, 8
      jr     $ra
L1:
    addi     $a0, $a0, -1
    jal      fact
    lw       $a0, 0($sp)
    lw       $ra, 4($sp)
    addi     $sp, $sp, 8
    mul      $v0, $a0, $v0
    jr       $ra
```

Memory Organization

- The space allocated on stack by a procedure is termed the activation record (includes saved values and data local to the procedure) – frame pointer points to the start of the record and stack pointer points to the end – variable addresses are specified relative to $fp as $sp may change during the execution of the procedure
- $gp points to area in memory that saves global variables
- Dynamically allocated storage (with malloc()) is placed on the heap



Procedure Calls

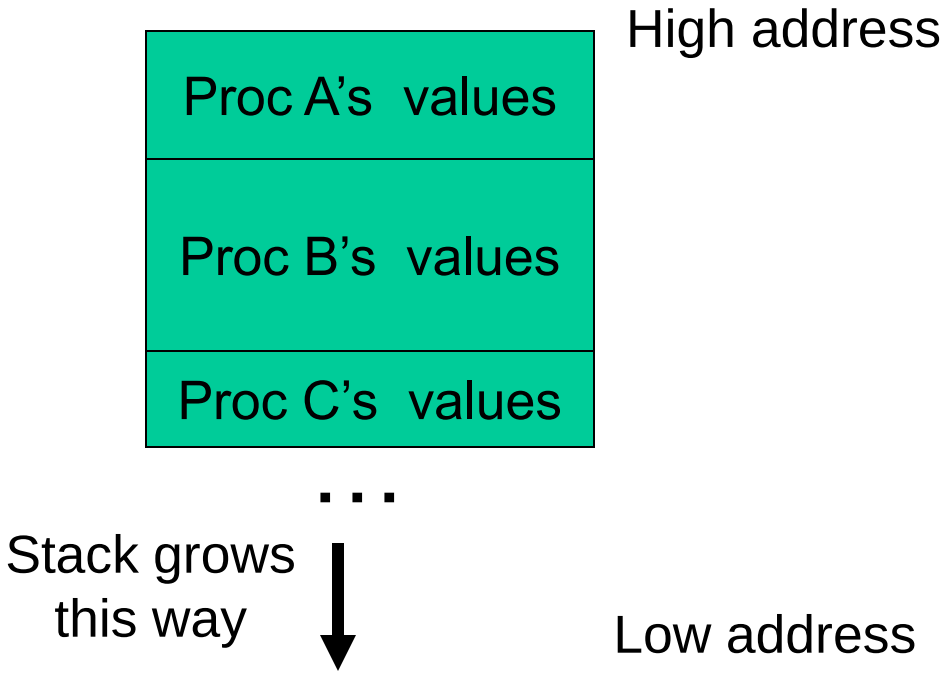
- The topics:
  - Procedure calls
  - Large constants
  - The compilation process
- Reminder:

Recap

- The jal instruction is used to jump to the procedure and save the current PC (+4) into the return address register
- Arguments are passed in $a0-$a3; return values in $v0-$v1
- Since the callee may over-write the caller's registers, relevant values may have to be copied into memory
- Each procedure may also require memory space for local variables – a stack is used to organize the memory needs for each procedure

The Stack

The register scratchpad for a procedure seems volatile – it seems to disappear every time we switch procedures – a procedure's values are therefore backed up in memory on a stack




```
Proc  A

        call  Proc B
              …
              call Proc C
                 …
              return
        return
return
```

Example 1

```
int  leaf_example (int g, int h, int i, int j)
{
    int f ;
    f = (g + h) – (i + j);
    return f;
}
```

Example 1

```
int  leaf_example (int g, int h, int i, int j)
{
    int f ;
    f = (g + h) – (i + j);
    return f;
}
```

```
leaf_example:
   addi     $sp,  $sp,  -12
   sw       $t1, 8($sp)
   sw       $t0, 4($sp)
   sw       $s0, 0($sp)
   add      $t0, $a0, $a1
   add      $t1, $a2, $a3
   sub      $s0, $t0, $t1
   add      $v0, $s0, $zero
   lw       $s0, 0($sp)
   lw       $t0, 4($sp)
   lw       $t1, 8($sp)
   addi     $sp, $sp, 12
   jr       $ra
```

Notes:
In this example, the procedure's stack space was used for the caller's variables, not the callee's – the compiler decided that was better.

The caller took care of saving its $ra and $a0-$a3.

Example 2

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

Example 2

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

Notes:
The caller saves $a0 and $ra in its stack space.
Temps are never saved.

```
fact:
    addi    $sp, $sp, -8
    sw      $ra, 4($sp)
    sw      $a0, 0($sp)
    slti    $t0, $a0, 1
    beq     $t0, $zero, L1
      addi  $v0, $zero, 1
      addi  $sp, $sp, 8
      jr    $ra
L1:
    addi    $a0, $a0, -1
    jal     fact
    lw      $a0, 0($sp)
    lw      $ra, 4($sp)
    addi    $sp, $sp, 8
    mul     $v0, $a0, $v0
    jr      $ra
```

Homework & Experiment

- Homework for Chapter 2 (p.149)
  - ☞2.6，2.15，2.37，2.46，2.47

- Experiment
    - ✧Every Friday evening 18:30~20:30
    - ✧Total 5 reports + a summarization
  - ☞1、C Programming for MIPS assemble, un-assemble, MIPS VM (Report before Chapter 5)
  - ☞2、ISE software,Basic I/O for Xilinx Spartan 3 (Report before next friday)

Experiments

周次	日期	教学内容	实践教学环节	时数
1	09.14	实验一：MIPS汇编语言的汇编 反汇编 MIPS模拟机	复习C语言编程	
2	09.21		手工编译C语言程序，MIPS汇编	
3	09.28		MIPS汇编、反汇编、MIPS模拟机	
4	10.12	实验二：ISE软件使用，实验板基本操作。基本I/O设计	ISE软件使用	
5	10.19		实验板基本I/O设计	
6	10.26	实验三：基本组件设计、ALU设计	MUX、译码器、寄存器	
7	11.02		寄存器组、存储器应用	
8	11.09		加法器、并行加法	
9	11.23		ALU	
10	11.30	实验四：单时钟CPU设计	组件的组合	
11	12.07		单时钟CPU设计	
12	12.14	实验五：多时钟CPU设计 1)PLA控制器 2)微程序控制	多时钟CPU设计	
13	12.21		PLA控制器	
14	12.28		微程序控制器设计	
15	01.04		微程序控制多时钟CPU设计	
16	01.11	实验总结	总共5+1个报告	

Experiments

Week	Date	Experiment	Content	Hrs.
1	09.14	1、C Programming 1)MIPS assembly, 2)un-assembly, 3)MIPS virtual machine	Review C Language	
2	09.21		C compiler, MIPS assemble	
3	09.28		un-assemble, MIPS VM	
4	10.12	2、ISE software, Basic I/O for Xilinx Spartan 3	ISE	
5	10.19		Xilinx Spartan 3	
6	10.26	3、Basic elements, ALU design	MUX、decode、register	
7	11.02		register file、memory	
8	11.09		Adder	
9	11.23		ALU design	
10	11.30	4、Single cycle datapath design	Element combination	
11	12.07		Single cycle CPU design	
12	12.14	5、Multi-cycle datapath design 1)PLA controller 2)Microprogramming	Multi-cycle CPU design	
13	12.21		PLA controller	
14	12.28		Microprogramming	
15	01.04		Microprogramming CPU	
16	01.11	Summarization	Total 5+1 reports	

Exercises

Convert to assembly:

```
int max (int *u, int k)
{
     int m=u[0];
     for(int i=1; i<k; i++)
        if(u[i]>m)m=u[i];
     return m;
}
```

Convert to assembly:

```
int strlen(char *u)
{
     int m=0;
     while(u[m]!=0)m++;
     return m;
}
```

Convert to assembly:

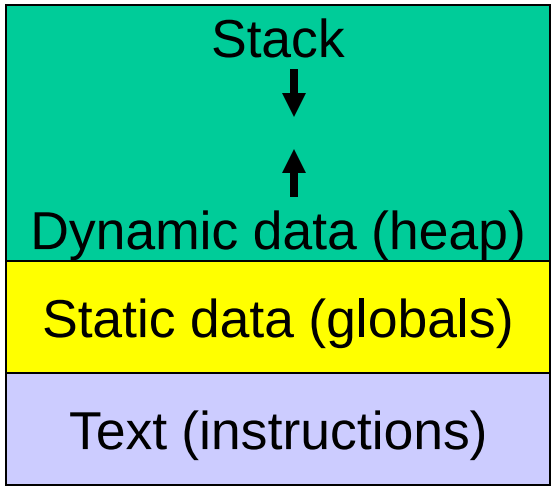
```
int sum (int *u, int k)
{
     int m=u[0];
     for(int i=1; i<k; i++)
        m+=u[i];
     return m;
}
```

Convert to assembly:

```
void mmax (int *u, int k)
{
   int m=max(k, u);
   for(int i=0; i<k; i++)
     u[i]=m;
}
```

Memory Organization

- The space allocated on stack by a procedure is termed the activation record (includes saved values and data local to the procedure) – frame pointer points to the start of the record and stack pointer points to the end – variable addresses are specified relative to $fp as $sp may change during the execution of the procedure
- $gp points to area in memory that saves global variables
- Dynamically allocated storage (with malloc()) is placed on the heap



Dealing with Characters

- Instructions are also provided to deal with byte-sized and half-word quantities: lb (load-byte), sb, lh, sh
- These data types are most useful when dealing with characters, pixel values, etc.
- C employs ASCII formats to represent characters – each character is represented with 8 bits and a string ends in the null character (corresponding to the 8-bit number 0)

Example

Convert to assembly:

```
int strcpy (char x[], char y[])
{
    int i;
    i=0;
    while  ((x[i] = y[i]) != `\0’)
    i += 1;
    rerun I;
}
```

Example

Convert to assembly:

```
int strcpy (char x[], char y[])
{
    int i;
    i=0;
    while  ((x[i] = y[i]) != `\0')
            i ++;
    return i
}
```

```
strcpy:
        addi    $sp, $sp, -4
        sw      $s0, 0($sp)
        add     $s0, $zero, $zero
L1:     add     $t1, $s0, $a1
        lb      $t2, 0($t1)
        add     $t3, $s0, $a0
        sb      $t2, 0($t3)
        beq     $t2, $zero, L2
        addi    $s0, $s0, 1
        j       L1
L2:     addi    $v0, $s0, 1
        lw      $s0, 0($sp)
        addi    $sp, $sp, 4
        jr      $ra
```

Jump Table

```
switch (k){
  case 0:
    ……(L0)
    break;
 case 1:
    ……(L1)
    break;
 case 2:
    ……(L2)
    break;
 case 3:
    ……(L3)
    break;
  default:
    ……(LL)
}
```

Jump Table

Jump Table
Switch address

```
switch (k){
  case 0:
L0: ……
    break;
 case 1:
L1: ……
    break;
 case 2:
L2: ……
    break;
 case 3:
L3: ……
    break;
  default:
LL: ……
}
Exit:  ……
```

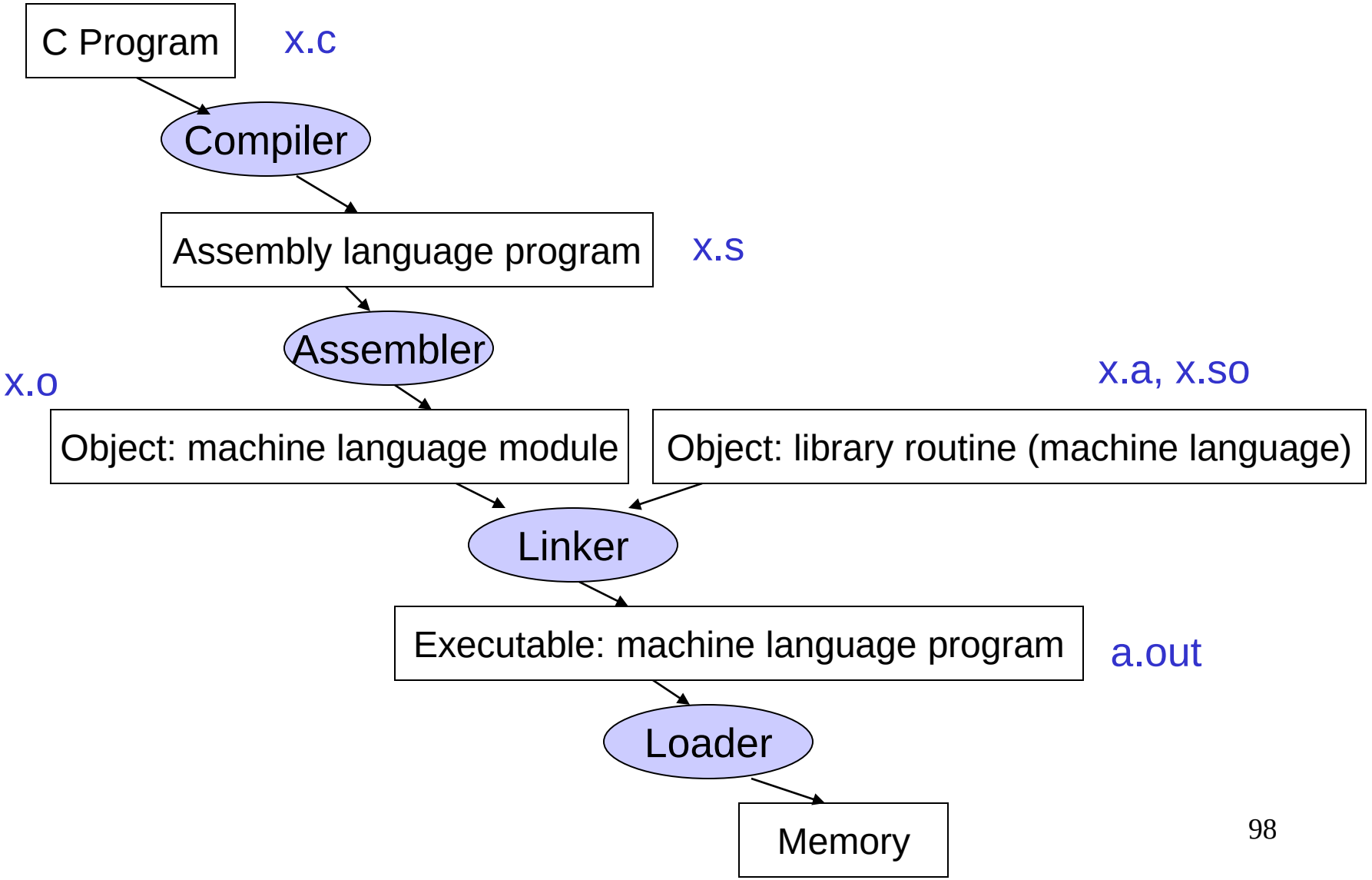
```
Switch: j       R0
        dw      L0
        dw      L1
        dw      L2
        dw      L3
        dw      LL
R0:     sltiu   $t0, $s0, 4
        bne     $t0, $zero, R1
        addi    $t0, $zero, 20
        j       R2
R1:     sll     $t0, $s0, 2
R2:     la      $t1, Switch
        add     $t0, $t0, $t1
        lw      $t0, 0($t0)
        jr      $t0
        j       Exit
L0:     ……
        j       Exit
L1:     ……
        j       Exit
L2:     ……
        j       Exit
L3:     ……
        j       Exit
LL:     ……
Exit:   ……
```

K→$s0

Large Constants

- Immediate instructions can only specify 16-bit constants
- The **lui** instruction is used to store a 16-bit constant into the upper 16 bits of a register… thus, two immediate instructions are used to specify a 32-bit constant
- The destination PC-address in a conditional branch is specified as a 16-bit constant, relative to the current PC
- A jump (j) instruction can specify a 26-bit constant; if more bits are required, the jump-register (jr) instruction is used

Starting a Program



Role of Assembler

- Convert pseudo-instructions into actual hardware instructions – pseudo-instrs make it easier to program in assembly – examples: “move”, “blt”, 32-bit immediate operands, etc.

- Convert assembly instrs into machine instrs – a separate object file (x.o) is created for each C file (x.c) – compute the actual values for instruction labels – maintain info on external references and debugging information

Role of Linker

- Stitches different object files into a single executable
  - patch internal and external references
  - determine addresses of data and instruction labels
  - organize code and data modules in memory
- Some libraries (DLLs) are dynamically linked – the executable points to dummy routines – these dummy routines call the dynamic linker-loader so they can update the executable to jump to the correct routine

Full Example – Sort in C

```
void sort (int v[], int n)
{
    int i, j;
    for (i=0; i<n; i+=1) {
        for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) {
            swap (v,j);
        }
    }
}
```

```
void swap (int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

- Allocate registers to program variables
- Produce code for the program body
- Preserve registers across procedure invocations

The swap Procedure

- Register allocation: $a0 and $a1 for the two arguments, $t0 for the temp variable – no need for saves and restores as we're not using $s0-$s7 and this is a leaf procedure (won't need to re-use $a0 and $a1)

```
swap:   sll   $t1, $a1, 2
        add   $t1, $a0, $t1
        lw    $t0, 0($t1)
        lw    $t2, 4($t1)
        sw    $t2, 0($t1)
        sw    $t0, 4($t1)
        jr    $ra
```

The sort Procedure

- Register allocation: arguments v and n use $a0 and $a1, i and j use $s0 and $s1; must save $a0 and $a1 before calling the leaf procedure

- The outer for loop looks like this: (note the use of pseudo-instrs)

```
                move   $s0, $zero          # initialize the loop
loopbody1: bge     $s0, $a1, exit1     # will eventually use slt and beq
                … body of inner loop …
                addi    $s0, $s0, 1
                j          loopbody1
exit1:
```

```
for (i=0; i<n; i+=1) {
   for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) {
         swap (v,j);
   }
}
```

The sort Procedure

- The inner for loop looks like this:

```
                addi    $s1, $s0, -1          # initialize the loop
loopbody2:      blt     $s1, $zero, exit2     # will eventually use slt and beq
                sll     $t1,  $s1, 2
                add     $t2, $a0, $t1
                lw      $t3, 0($t2)
                lw      $t4, 4($t2)
                bgt     $t3, $t4, exit2
                … body of inner loop …
                addi    $s1, $s1, -1
                j        loopbody2
exit2:
```

```
for (i=0; i<n; i+=1) {
   for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) {
         swap (v,j);
   }
}
```

Saves and Restores

- Since we repeatedly call “swap” with $a0 and $a1, we begin “sort” by copying its arguments into $s2 and $s3 – must update the rest of the code in “sort” to use $s2 and $s3 instead of $a0 and $a1
- Must save $ra at the start of “sort” because it will get over-written when we call “swap”
- Must also save $s0-$s3 so we don’t overwrite something that belongs to the procedure that called “sort”

Saves and Restores

```
sort:   addi    $sp, $sp, -20
        sw      $ra, 16($sp)
        sw      $s3, 12($sp)
        sw      $s2, 8($sp)
        sw      $s1, 4($sp)
        sw      $s0, 0($sp)
        move    $s2, $a0
        move    $s3, $a1
          …
        move    $a0, $s2        # the inner loop body starts here
        move    $a1, $s1
        jal     swap
          …
exit1:  lw      $s0, 0($sp)
          …
        addi    $sp, $sp, 20
        jr      $ra
```

9 lines of C code → 35 lines of assembly

Relative Performance

Gcc optimization	Relative performance	Cycles	Instruction count	CPI
none	1.00	159B	115B	1.38
O1	2.37	67B	37B	1.79
O2	2.38	67B	40B	1.66
O3	2.41	66B	45B	1.46

- A Java interpreter has relative performance of 0.12, while the Jave just-in-time compiler has relative performance of 2.13
- Note that the quicksort algorithm is about three orders of magnitude faster than the bubble sort algorithm (for 100K elements)

Lecture 5: MIPS Examples

- Today's topics:
  - the compilation process
  - full example – sort in C
- Reminder: 2nd assignment will be posted later today

Dealing with Characters

- Instructions are also provided to deal with byte-sized and half-word quantities: lb (load-byte), sb, lh, sh
- These data types are most useful when dealing with characters, pixel values, etc.
- C employs ASCII formats to represent characters – each character is represented with 8 bits and a string ends in the null character (corresponding to the 8-bit number 0)

Example

Convert to assembly:

```
void strcpy (char x[], char y[])
{
    int i;
    i=0;
    while  ((x[i] = y[i]) != `\0’)
    i += 1;
}
```

Example

Convert to assembly:

```
void strcpy (char x[], char y[])
{
    int i;
    i=0;
    while  ((x[i] = y[i]) != `\0’)
       i += 1;
}
```

```
strcpy:
 addi    $sp, $sp, -4
 sw      $s0, 0($sp)
 add     $s0, $zero, $zero
L1: add  $t1, $s0, $a1
 lb      $t2, 0($t1)
 add     $t3, $s0, $a0
 sb      $t2, 0($t3)
 beq     $t2, $zero, L2
 addi    $s0, $s0, 1
 j       L1
L2: lw   $s0, 0($sp)
 addi    $sp, $sp, 4
 jr      $ra
```

Large Constants

- Immediate instructions can only specify 16-bit constants
- The lui instruction is used to store a 16-bit constant into the upper 16 bits of a register… thus, two immediate instructions are used to specify a 32-bit constant
- The destination PC-address in a conditional branch is specified as a 16-bit constant, relative to the current PC
- A jump (j) instruction can specify a 26-bit constant; if more bits are required, the jump-register (jr) instruction is used

Role of Assembler

- Convert pseudo-instructions into actual hardware instructions – pseudo-instrs make it easier to program in assembly – examples: “move”, “blt”, 32-bit immediate operands, etc.

- Convert assembly instrs into machine instrs – a separate object file (x.o) is created for each C file (x.c) – compute the actual values for instruction labels – maintain info on external references and debugging information

Role of Linker

- Stitches different object files into a single executable
  - patch internal and external references
  - determine addresses of data and instruction labels
  - organize code and data modules in memory
- Some libraries (DLLs) are dynamically linked – the executable points to dummy routines – these dummy routines call the dynamic linker-loader so they can update the executable to jump to the correct routine

Full Example – Sort in C

```
void sort (int v[], int n)
{
    int i, j;
    for (i=0; i<n; i+=1) {
        for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) {
            swap (v,j);
        }
    }
}
```

```
void swap (int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

- Allocate registers to program variables
- Produce code for the program body
- Preserve registers across procedure invocations

The swap Procedure

```
void swap (int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

- Allocate registers to program variables
- Produce code for the program body
- Preserve registers across procedure invocations

The swap Procedure

- Register allocation: $a0 and $a1 for the two arguments, $t0 for the temp variable – no need for saves and restores as we're not using $s0-$s7 and this is a leaf procedure (won't need to re-use $a0 and $a1)

```
swap:   sll   $t1, $a1, 2
        add   $t1, $a0, $t1
        lw    $t0, 0($t1)
        lw    $t2, 4($t1)
        sw    $t2, 0($t1)
        sw    $t0, 4($t1)
        jr    $ra
```

```
void swap (int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

The sort Procedure

- Register allocation: arguments v and n use $a0 and $a1, i and j use $s0 and $s1

```
for (i=0; i<n; i+=1) {
   for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) {
         swap (v,j);
   }
}
```

The sort Procedure

- Register allocation: arguments v and n use $a0 and $a1, i and j use $s0 and $s1; must save $a0, $a1, and $ra before calling the leaf procedure

- The outer for loop looks like this: (note the use of pseudo-instrs)

```
                move   $s0, $zero          # initialize the loop
loopbody1: bge     $s0, $a1, exit1     # will eventually use slt and beq
                … body of inner loop …
                addi    $s0, $s0, 1
                j          loopbody1
exit1:
```

```
for (i=0; i<n; i+=1) {
   for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) {
         swap (v,j);
   }
}
```

The sort Procedure

- The inner for loop looks like this:

```
                addi    $s1, $s0, -1          # initialize the loop
loopbody2:      blt     $s1, $zero, exit2     # will eventually use slt and beq
                sll     $t1,  $s1, 2
                add     $t2, $a0, $t1
                lw      $t3, 0($t2)
                lw      $t4, 4($t2)
                bge     $t4, $t3, exit2
                … body of inner loop …
                addi    $s1, $s1, -1
                j        loopbody2
exit2:
```

```
for (i=0; i<n; i+=1) {
   for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) {
         swap (v,j);
   }
}
```

Saves and Restores

- Since we repeatedly call “swap” with $a0 and $a1, we begin “sort” by copying its arguments into $s2 and $s3 – must update the rest of the code in “sort” to use $s2 and $s3 instead of $a0 and $a1
- Must save $ra at the start of “sort” because it will get over-written when we call “swap”
- Must also save $s0-$s3 so we don’t overwrite something that belongs to the procedure that called “sort”

Saves and Restores

```
sort:   addi    $sp, $sp, -20
        sw      $ra, 16($sp)
        sw      $s3, 12($sp)
        sw      $s2, 8($sp)
        sw      $s1, 4($sp)
        sw      $s0, 0($sp)
        move    $s2, $a0
        move    $s3, $a1
          …
        move    $a0, $s2        # the inner loop body starts here
        move    $a1, $s1
        jal     swap
          …
exit1:  lw      $s0, 0($sp)
          …
        addi    $sp, $sp, 20
        jr      $ra
```

9 lines of C code → 35 lines of assembly

```
for (i=0; i<n; i+=1) {
   for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) {
         swap (v,j);
   }
}
```

Relative Performance

Gcc optimization	Relative performance	Cycles	Instruction count	CPI
none	1.00	159B	115B	1.38
O1	2.37	67B	37B	1.79
O2	2.38	67B	40B	1.66
O3	2.41	66B	45B	1.46

- A Java interpreter has relative performance of 0.12, while the Jave just-in-time compiler has relative performance of 2.13
- Note that the quicksort algorithm is about three orders of magnitude faster than the bubble sort algorithm (for 100K elements)

IA-32 Instruction Set

- Intel's IA-32 instruction set has evolved over 20 years – old features are preserved for software compatibility
- Numerous complex instructions – complicates hardware design (Complex Instruction Set Computer – CISC)
- Instructions have different sizes, operands can be in registers or memory, only 8 general-purpose registers, one of the operands is over-written
- RISC instructions are more amenable to high performance (clock speed and parallelism) – modern Intel processors convert IA-32 instructions into simpler micro-operations

Lecture 6: Compilers, the SPIM Simulator

- Today's topics:
  - SPIM simulator
  - The compilation process

- Additional TA hours:
  Liqun Cheng, email legion at cs, Office: MEB 2162
  Office hours: Mon/Wed 11-12

  TA hours for Josh: Wed 11:45-12:45 (EMCB 130)
  TA hours for Devyani: Wed 11:45-12:45 (MEB 3431)

IA-32 Instruction Set

- Intel's IA-32 instruction set has evolved over 20 years – old features are preserved for software compatibility
- Numerous complex instructions – complicates hardware design (Complex Instruction Set Computer – CISC)
- Instructions have different sizes, operands can be in registers or memory, only 8 general-purpose registers, one of the operands is over-written
- RISC instructions are more amenable to high performance (clock speed and parallelism) – modern Intel processors convert IA-32 instructions into simpler micro-operations

SPIM

- SPIM is a simulator that reads in an assembly program and models its behavior on a MIPS processor
- Note that a “MIPS add instruction” will eventually be converted to an add instruction for the host computer’s architecture – this translation happens under the hood
- To simplify the programmer’s task, it accepts pseudo-instructions, large constants, constants in decimal/hex formats, labels, etc.
- The simulator allows us to inspect register/memory values to confirm that our program is behaving correctly

Example

This simple program (similar to what we've written in class) will run on SPIM (a "main" label is introduced so SPIM knows where to start)

```
main:
    addi   $t0, $zero, 5
    addi   $t1, $zero, 7
    add    $t2, $t0, $t1
```

If we inspect the contents of $t2, we'll find the number 12

User Interface

```
rajeev@trust > spim

(spim)  read “add.s”
(spim)  run
(spim)  print $10
  Reg 10 = 0x0000000c (12)
(spim)  reinitialize
(spim)  read “add.s”
(spim)  step
(spim)  print $8
  Reg 8 = 0x00000005  (5)
(spim)  print $9
  Reg 9 = 0x00000000  (0)
(spim)  step
(spim)  print $9
  Reg 9 = 0x00000007  (7)
(spim)  exit
```

File add.s

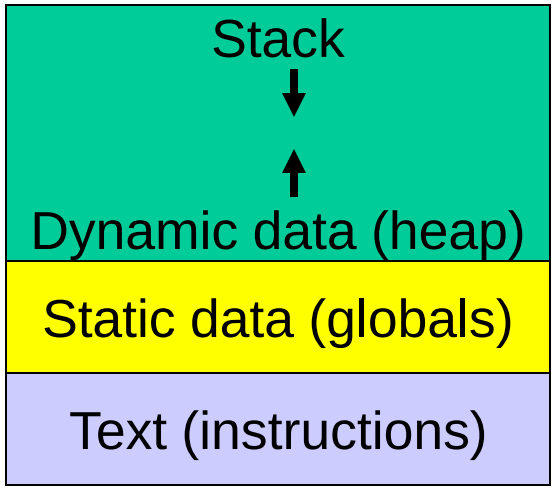
```
main:
    addi   $t0, $zero, 5
    addi   $t1, $zero, 7
    add    $t2, $t0, $t1
```

Directives

File add.s

```
.text
.globl main
main:
    addi   $t0, $zero, 5
    addi   $t1, $zero, 7
    add    $t2, $t0, $t1
    …
    jal      swap_proc
    jr       $ra

.globl swap_proc
swap_proc:
     …
```

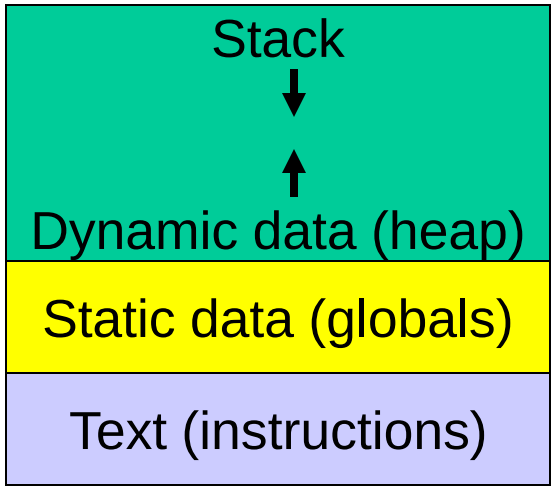


This function is visible to other files

Directives

File add.s

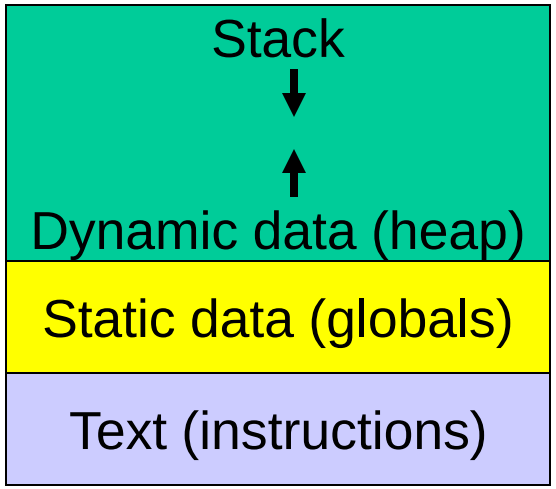
```
.data
   .word 5
   .word 7
   .byte  25
   .asciiz  “the answer is”
.text
.globl main
main:
    lw      $t0, 0($gp)
    lw      $t1, 4($gp)
    add    $t2, $t0, $t1
    …
    jal      swap_proc
    jr       $ra
```



Labels

File add.s

```
.data
in1   .word 5
in2   .word 7
c1    .byte  25
str    .asciiz  “the answer is”
.text
.globl main
main:
    lw       $t0, in1
    lw       $t1, in2
    add     $t2, $t0, $t1
    …
    jal       swap_proc
    jr       $ra
```



Endian-ness

Two major formats for transferring values between registers and memory

Memory: low address 45 7b 87 7f high address

Little-endian register: the first byte read goes in the low end of the register

Register: 7f 87 7b 45

Most-significant bit ↗ ↖ Least-significant bit

Big-endian register: the first byte read goes in the big end of the register

Register: 45 7b 87 7f

Most-significant bit ↗ ↖ Least-significant bit

System Calls

- SPIM provides some OS services: most useful are operations for I/O: read, write, file open, file close
- The arguments for the syscall are placed in $a0-$a3
- The type of syscall is identified by placing the appropriate number in $v0 – 1 for print_int, 4 for print_string, 5 for read_int, etc.
- $v0 is also used for the syscall's return value

Example Print Routine

```
.data
   str:     .asciiz   “the answer is ”
.text
   li       $v0, 4              # load immediate; 4 is the code for print_string
   la       $a0, str            #  the print_string syscall expects the string
                                #  address as the argument; la is the instruction
                                #  to load the address of the operand (str)
   syscall                      #  SPIM will now invoke syscall-4
   li       $v0, 1              #  syscall-1 corresponds to print_int
   li       $a0, 5              #  print_int expects the integer as its argument
   syscall                      #  SPIM will now invoke syscall-1
```

Example

- Write an assembly program to prompt the user for two numbers and print the sum of the two numbers

Example

```
.text
   .globl main
   main:
      li   $v0, 4
      la  $a0, str1
      syscall
      li   $v0, 5
      syscall
      add  $t0, $v0, $zero
      li   $v0, 5
      syscall
      add  $t1, $v0, $zero
      li   $v0, 4
      la  $a0, str2
      syscall
      li    $v0, 1
      add  $a0, $t1, $t0
      syscall
```

```
.data
   str1:  .asciiz  “Enter 2 numbers:”
   str2:  .asciiz  “The sum is ”
```

Compilation Steps

- The front-end: deals mostly with language specific actions
  - Scanning: reads characters and breaks them into tokens
  - Parsing: checks syntax
  - Semantic analysis: makes sure operations/types are meaningful
  - Intermediate representation: simple instructions, infinite registers, makes few assumptions about hw
- The back-end: optimizations and code generation
  - Local optimizations: within a basic block
  - Global optimizations: across basic blocks
  - Register allocation

Dataflow

- Control flow graph: each box represents a basic block and arcs represent potential jumps between instructions
- For each block, the compiler computes values that were defined (written to) and used (read from)
- Such dataflow analysis is key to several optimizations: for example, moving code around, eliminating dead code, removing redundant computations, etc.

Register Allocation

- The IR contains infinite virtual registers – these must be mapped to the architecture's finite set of registers (say, 32 registers)
- For each virtual register, its live range is computed (the range between which the register is defined and used)
- We must now assign one of 32 colors to each virtual register so that intersecting live ranges are colored differently – can be mapped to the famous graph coloring problem
- If this is not possible, some values will have to be temporarily spilled to memory and restored (this is equivalent to breaking a single live range into smaller live ranges)

High-Level Optimizations

High-level optimizations are usually hardware independent

- Procedure inlining
- Loop unrolling
- Loop interchange, blocking (more on this later when we study cache/memory organization)

Low-Level Optimizations

- Common sub-expression elimination
- Constant propagation
- Copy propagation
- Dead store/code elimination
- Code motion
- Induction variable elimination
- Strength reduction
- Pipeline scheduling

Title

- Bullet