MIPS Assembly

Operating Systems (CST 1A)
Michaelmas 2008
Handout 2

What is MIPS?

- A Reduced Instruction Set Computer (RISC) microprocessor:
  - Developed at Stanford in the 1980s [Hennessy]
  - Designed to be fast and simple
  - Originally 32-bit; today also get 64-bit versions
  - Primarily used in embedded systems (e.g. routers, TiVo’s, PSPs…)
  - First was R2000 (1985); later R3000, R4000, …
- Also used by big-iron SGI machines (R1x000)

MIPS Instructions

- MIPS has 3 instruction formats:
  - R-type - register operands
  - I-type - immediate operands
  - J-type - jump operands
- All instructions are 1 word long (32 bits)
- Examples of R-type instructions:

```
add     $8, $1, $2      # $8 <= $1 + $2
sub     $12, $6, $3     # $12 <= $6 - $3
and     $1, $2, $3      # $1 <= $2 & $3
or      $1, $2, $3      # $1 <= $2 | $3
```

- Register 0 ($0) always contains zero

```
add     $8, $0, $0      # $8 <= 0
add     $8, $1, $0      # $8 <= $1
```

R-Type Instructions

- Consists of six fixed-width fields:

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

op: basic operation of the instruction, typically called the opcode

rs: the first register source operand

rt: the second register source operand

rd: the register destination operand; gets result of the operation

shamt: shift amount (0 if not shift instruction)

funct: function. This field selects the specific variant of the operation and is sometimes called the *function code*; e.g. for op = 0: (funct = 32) => add ; (funct = 34) => sub

I-Type Instructions

op	rs	rt	address
6 bits	5 bits	5 bits	16 bits

- I = immediate
- Useful for loading constants, e.g:
  - li $7, 12   # load constant 12 into reg $7
- Opcode determines the format
- Also used for various other instructions…

Immediate Addressing on MIPS

- **or, and, xor** and **add** instructions have immediate forms (ori, andi, xori and addi), e.g.

```
ori     $8, $0, 0x123       # puts 0x0000 0123 into reg 8
ori     $9, $0, -6          # puts 0x0000 fffa into reg 9
addi    $10, $0, 0x123      # puts 0x0000 0123 into reg 10
addi    $11, $0, -6         # puts 0xffff fffa into reg 11
                            # note sign extension...
```

- **lui** instruction loads upper 16 bits with constant and sets ls 16 bits to zero

```
lui     $8, 0xabcd          # puts 0xabcd 0000 into reg 8
ori     $8, $0, 0x123       # sets ls bits; (reg 8 = 0xabcd 0123)
```

- **li** *pseudo-instruction* (see later) generates lui/ori or ori code sequence as needed...

J-Type Instruction

- Last instruction format - Jump-type (J-type)

opcode	Target address [26 bits]

- Only used by unconditional jumps, e.g.
  - `j dest_addr       # jump to (target<<2)`
  - Cannot directly jump more than 2^{26} instructions away…
- *Branches* use I-type, not J-type:
  - Must specify 2 registers to compare, e.g.
  - `beq $1, $2, dest  # goto dest iff $1==$2`
  - 16 bit offset => must be within 2^{16} instructions

Big Picture

High level Language

x = a - b + c - d;

Assembly

```
sub     $10, $4, $5
sub     $11, $6, $7
add     $12, $10, $11
```

0	4	5	10	0	34
0	6	7	11	0	34
0	10	11	12	0	32

Machine

```
000000 00100 00101 01010 00000 100010
000000 00110 00111 01011 00000 100010
000000 01010 01011 01100 00000 100000
```

Assumes that a, b, c, d are in $4, $5, $6, $7 somehow

MIPS Register Names

- Registers are used for specific purposes, by *convention*
- For example, register 4, 5, 6 and 7 are used as parameters or arguments for subroutines (see later)
- They can be specified as $4, $5, $6, $7 or as $a0, $a1, $a2 and $a4
- Other examples:

$zero	$0	zero
$at	$1	assembler temporary
$v0, $v1	$2, $3	expression evaluation & result
$t0 ... $t7	$8 ... $15	temporary registgers
$s0 ... $s7	$16 ... $23	saved temporaries
$t8, $t9	$24, $25	temporary
$k0, $k1	$26, $27	kernel temporaries
$gp	$28	global pointer
$sp	$29	stack pointer
$fp	$30	frame pointer
$ra	$31	return address

Our first program: Hello World!

```
        .text                      # code section
        .globl main
main:   li $v0, 4                  # system call for print string
        la $a0, str                # load address of string to print
        syscall                    # print the string
        li $v0, 10                 # system call for exit
        syscall                    # exit
        .data
str:    .asciiz “Hello world!\n”    # NUL terminated string, as in C
```

- Comments to aid readability
- Assembly language 5-20x line count of high level languages
- Development time strongly related to number of lines of code

Assembler Directives

- To assist assembler to do its job ...
- ... but do not necessarily produce results in memory
- Examples:

.text		tells assembler that following is part of code area
.data		following is part of data area
.ascii	**str**	insert Ascii string into next few bytes of memory
.asciiz	**str**	as above, but add null byte at end
.word	**n1,n2**	reserve space for words and store values n1, n2 etc. in them
.half	**n1,n2**	reserve space for halfwords and store values n1, n2 etc. in them
.byte	**n1,n2**	reserve space for bytes and store values n1, n2 etc. in them
.space	**n**	reserve space for n bytes
.align	**m**	align the next datum on 2^m byte boundary, e.g. .align 2 aligns on word boundary

.text and .data Directives

- Can be many of these in a program
- Unless specified, successive areas of each type are concatenated. The following are equivalent:

```
        .text
        add     ...
        sub     ...
        .data
s1:     .ascii  “abc”
        .text
        jal     ...
        jr      ...
        .data
s2:     .ascii  “xyz”
```

```
        .text
        add     ...
        sub     ...
        jal     ...
        jr      ...
        .data
s1:     .ascii  “abc”
s2:     .ascii  “xyz”
```

Pseudo Instructions

- Assembler may assist by providing *pseudo-instructions* which do not exist in real machine but can be built from others.
- Some examples are:

Pseudo Instructions		Translated to:	
move	$1,$2	add	$1, $0, $2
li	$1, 678	ori	$1, $0, 678
la	$8, 6($1)	addi	$8, $1, 6
la	$8, label	lui	$1, [label-hi]
		ori	$8, $1, [label-lo]
b	label	bgez	$0, $0, label
beq	$8, 66, label	ori	$1, $0, 66
		beq	$1, $8, label

Load and move instructions

```
la  $a0, addr        # load address addr into $a0
li   $a0, 12         # load immediate $a0 = 12
lb $a0, c($s1)       # load byte $a0 = Mem[$s1+c]
lh $a0, c($s1)       # load half word [16-bits]
lw $a0, c($s1)       # load word [32-bits]
move $s0, $s1        # $s0 = $s1
```

Control Flow Instructions

Assembly language has very few control structures:

- Branch instructions: if <cond> then goto <label>

```
beqz $s0, label          # if $s0==0      goto label
bnez $s0, label          # if $s0!=0      goto label
bge  $s0, $s1, label     # if $s0>=$s1    goto label
ble  $s0, $s1, label     # if $s0<=$s1    goto label
blt  $s0, $s1, label     # if $s0<$s1     goto label
beq  $s0, $s1, label     # if $s0==$s1    goto label
bgez  $s0, $s1, label    # if $s0>=0      goto label
```

- Jump instructions: goto label

We can build while loops, for loops, repeat-until loops, if-then-else structures from these primitives

If-then-else

if ($t0==$t1) then /* blockA */ else /* blockB */

```
   beq $t0, $t1, blockA
   j blockB
blockA: … instructions of then block …
   j exit
blockB: … instructions of else block …
exit:  … subsequent instructions …
```

Repeat-Until

repeat … until $t0>$t1

```
   … initialize $t0 …
loop: … instructions of loop …
   sub $t0, $t0, 1        # decrement $t0
   ble $t0, $t1, loop     # if $t0<=$t1 goto loop
```

Other loop structures are similar…

Jump Instructions

- *J-type* instructions have *6 bit opcode* and *26 bit address*
- Larger range of transfer addresses
- Also specifies *word* address, not byte address, so effectively 2^{28} byte addressing
- Assembler converts very distant conditional branches to inverse-branch and jump

```
        beq     $3, $2, very_distant_label
```

- converted to:

```
        bne     $3, $2, label1          # continue
        j       very_distant_label
lbl1:   ... instructions ...            # continue from here
```

MIPS Indirect Jump

- Indirect jump via register (32 bit address) too, e.g.

```
            jr      $4      # reg 4 contains target address
```

- Also can be used for *jump table*
- Suppose we need to branch to different locations depending on value in register 5, e.g. 0, 4, 8, 12, 16, 20

```
            .data
label:      .word   l1, l2, l3, l4, l5, l6
            .text
main:       ...                     # instructions setting reg 4
            lw      $10, label($4)
            jr      $10
l1:                 ...
l2:                 ...             # and other labels too
```

Simulators

- Any digital computer can, in principle, be programmed to simulate any other
- For example, SPIM simulator can run on various machines including Unix, PC/Windows, and Mac
- Simulators may only simulate part of machine, e.g. only some input/output devices, a subset of instructions etc.
- Simulators are generally slower than using the real machine, but e.g. simulating 1960's machine on modern hardware may be faster than original

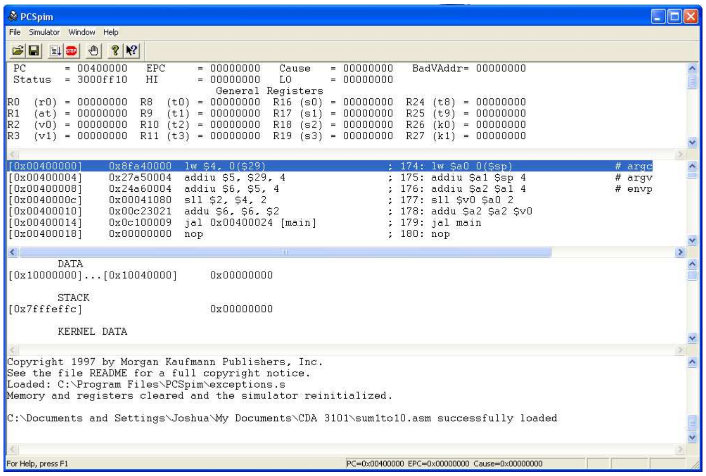
Interpreters and Abstract Machines

- Simulation generally refers to reproducing effects of a real machine ...
- ... but can design an abstract or virtual machine which is never implemented in hardware
- Often used for portability with compilers, e.g. p-code machine for Pascal, or Java Virtual Machine for Java, etc.
- An interpreter is then used to execute the code. The interpreter can easily be implemented on a variety of machines as a conventional program and takes p-code or JVM-code as data (instructions)
- Interpreters and simulators are identical concepts

Spim Simulator

- *"$^{1}/_{25}$th the performance at none of the cost"*
- Simulates a MIPS-based machine
- Includes some basic input/output routines to make programming easier
- Installation
  1. From the Patterson & Hennesey textbook CD
  2. From the internet
     http://www.cs.wisc.edu/~larus/spim.html

PC Spim



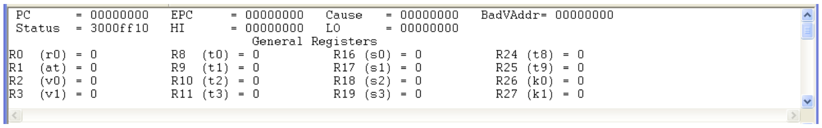
```
 PC       = 00400000   EPC      = 00000000   Cause    = 00000000   BadVAddr= 00000000
 Status   = 3000ff10   HI       = 00000000   LO       = 00000000
                                General Registers
R0  (r0) = 00000000  R8  (t0) = 00000000  R16 (s0) = 00000000  R24 (t8) = 00000000
R1  (at) = 00000000  R9  (t1) = 00000000  R17 (s1) = 00000000  R25 (t9) = 00000000
R2  (v0) = 00000000  R10 (t2) = 00000000  R18 (s2) = 00000000  R26 (k0) = 00000000
R3  (v1) = 00000000  R11 (t3) = 00000000  R19 (s3) = 00000000  R27 (k1) = 00000000
```

```
[0x00400000]    0x8fa40000  lw $4, 0($29)                   ; 174: lw $a0 0($sp)               # argc
[0x00400004]    0x27a50004  addiu $5, $29, 4                ; 175: addiu $a1 $sp 4             # argv
[0x00400008]    0x24a60004  addiu $6, $5, 4                 ; 176: addiu $a2 $a1 4             # envp
[0x0040000c]    0x00041080  sll $2, $4, 2                   ; 177: sll $v0 $a0 2
[0x00400010]    0x00c23021  addu $6, $6, $2                 ; 178: addu $a2 $a2 $v0
[0x00400014]    0x0c100009  jal 0x00400024 [main]           ; 179: jal main
[0x00400018]    0x00000000  nop                             ; 180: nop
```

```
        DATA
[0x10000000]...[0x10040000]     0x00000000

        STACK
[0x7fffeffc]                    0x00000000

        KERNEL DATA
```

```
Copyright 1997 by Morgan Kaufmann Publishers, Inc.
See the file README for a full copyright notice.
Loaded: C:\Program Files\PCSpim\exceptions.s
Memory and registers cleared and the simulator reinitialized.

C:\Documents and Settings\Joshua\My Documents\CDA 3101\sum1to10.asm successfully loaded
```

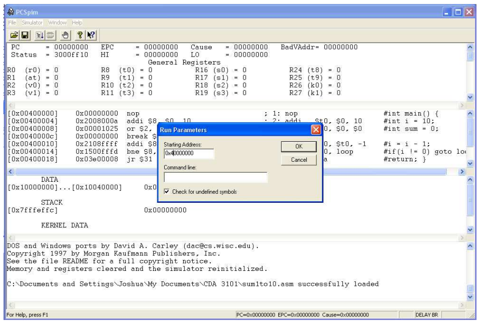
PC Spim



```
 PC       = 00000000   EPC      = 00000000   Cause    = 00000000   BadVAddr= 00000000
 Status   = 3000ff10   HI       = 00000000   LO       = 00000000
                                General Registers
R0  (r0) = 0              R8  (t0) = 0              R16 (s0) = 0              R24 (t8) = 0
R1  (at) = 0              R9  (t1) = 0              R17 (s1) = 0              R25 (t9) = 0
R2  (v0) = 0              R10 (t2) = 0              R18 (s2) = 0              R26 (k0) = 0
R3  (v1) = 0              R11 (t3) = 0              R19 (s3) = 0              R27 (k1) = 0
```

- Note the top window – it contains the state of all registers.

PC Spim



```
 PC       = 00000000   EPC      = 00000000   Cause    = 00000000   BadVAddr= 00000000
 Status   = 3000ff10   HI       = 00000000   LO       = 00000000
                                 General Registers
R0  (r0) = 0           R8  (t0) = 0           R16 (s0) = 0           R24 (t8) = 0
R1  (at) = 0           R9  (t1) = 0           R17 (s1) = 0           R25 (t9) = 0
R2  (v0) = 0           R10 (t2) = 0           R18 (s2) = 0           R26 (k0) = 0
R3  (v1) = 0           R11 (t3) = 0           R19 (s3) = 0           R27 (k1) = 0

[0x00400000]     0x00000000  nop                          ; 1: nop                    #int main() {
[0x00400004]     0x2008000a  addi $8, $0, 10              ; 2: addi   $t0, $0, 10     #int i = 10;
[0x00400008]     0x00001025  or $2,                            0, $0, $0              #int sum = 0;
[0x0040000c]     0x00000000  break $
[0x00400010]     0x2108ffff  addi $8                           0, $t0, -1             #i = i - 1;
[0x00400014]     0x1500fffd  bne $8,                           0, loop                #if(i != 0) goto lo
[0x00400018]     0x03e00008  jr $31                            a                      #return; }

        DATA
[0x10000000]...[0x10040000]     0x0

        STACK
[0x7fffeffc]                    0x00000000

        KERNEL DATA

DOS and Windows ports by David A. Carley (dac@cs.wisc.edu).
Copyright 1997 by Morgan Kaufmann Publishers, Inc.
See the file README for a full copyright notice.
Memory and registers cleared and the simulator reinitialized.

C:\Documents and Settings\Joshua\My Documents\CDA 3101\sum1to10.asm successfully loaded
```

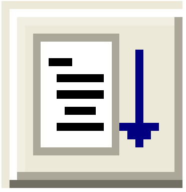
Starting Address: 0x400000

Command line:

☑ Check for undefined symbols

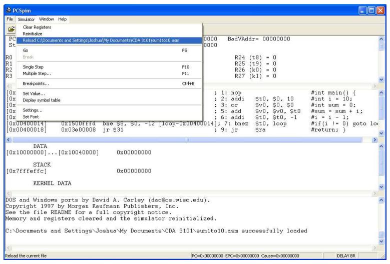
For Help, press F1

PC=0x00000000 EPC=0x00000000 Cause=0x00000000 DELAY BR



- The button on the top right runs the program to its end, after you click "OK" on the dialog box. Note that you won't see the register changes until the program ends.

PC Spim



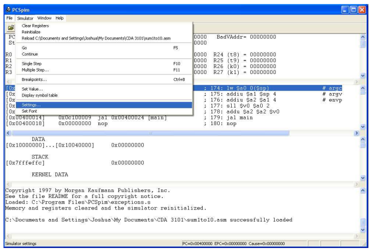
```
PC                                                   0000   BadVAddr= 00000000
St                                                   0000

R0                                                           R24 (t8) = 0
R1                                                           R25 (t9) = 0
R2                                                           R26 (k0) = 0
R3                                                           R27 (k1) = 0

[0x                                                  ; 1: nop                    #int main() {
[0x                                                  ; 2: addi   $t0, $0, 10     #int i = 10;
[0x                                                  ; 3: or     $v0, $0, $0     #int sum = 0;
[0x                                                  ; 5: add    $v0, $v0, $t0   #sum = sum + i;
[0x                                                  ; 6: addi   $t0, $t0, -1    #i = i - 1;
[0x00400014]     0x1500fffd  bne $8, $0, -12 [loop-0x00400014]; 7: bnez   $t0, loop     #if(i != 0) goto lo
[0x00400018]     0x03e00008  jr $31                  ; 9: jr     $ra             #return; }

        DATA
[0x10000000]...[0x10040000]     0x00000000

        STACK
[0x7fffeffc]                    0x00000000

        KERNEL DATA

DOS and Windows ports by David A. Carley (dac@cs.wisc.edu).
Copyright 1997 by Morgan Kaufmann Publishers, Inc.
See the file README for a full copyright notice.
Memory and registers cleared and the simulator reinitialized.

C:\Documents and Settings\Joshua\My Documents\CDA 3101\sum1to10.asm successfully loaded
```

Reload the current file

PC=0x00000000 EPC=0x00000000 Cause=0x00000000 DELAY BR

- Click this menu item to reinitialize PC Spim – it's like rebooting your computer. It's often necessary to click Reload to run your program again.

PC Spim

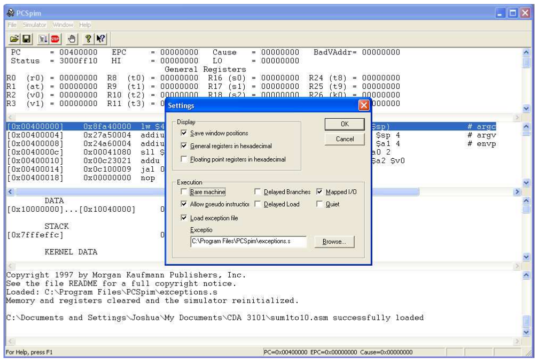


```
PC                                      0000   BadVAddr= 00000000
St                                      0000

R0                                      0000  R24 (t8) = 00000000
R1                                      0000  R25 (t9) = 00000000
R2                                      0000  R26 (k0) = 00000000
R3                                      0000  R27 (k1) = 00000000

[0x                                     ; 174: lw $a0 0($sp)            # argc
[0x                                     ; 175: addiu $a1 $sp 4          # argv
[0x                                     ; 176: addiu $a2 $a1 4          # envp
[0x                                     ; 177: sll $v0 $a0 2
[0x                                     ; 178: addu $a2 $a2 $v0
[0x00400014]    0x0c100009  jal 0x00400024 [main]     ; 179: jal main
[0x00400018]    0x00000000  nop                       ; 180: nop

        DATA
[0x10000000]...[0x10040000]     0x00000000

        STACK
[0x7fffeffc]                    0x00000000

        KERNEL DATA

Copyright 1997 by Morgan Kaufmann Publishers, Inc.
See the file README for a full copyright notice.
Loaded: C:\Program Files\PCSpim\exceptions.s
Memory and registers cleared and the simulator reinitialized.

C:\Documents and Settings\Joshua\My Documents\CDA 3101\sum1to10.asm successfully loaded
```

PC=0x00400000 EPC=0x00000000 Cause=0x00000000

- Click this menu item to change settings for the emulator.

PC Spim



```
PC      = 00400000   EPC     = 00000000   Cause    = 00000000   BadVAddr= 00000000
Status  = 3000ff10   HI      = 00000000   LO       = 00000000
                              General Registers
R0  (r0) = 00000000  R8  (t0) = 00000000  R16 (s0) = 00000000  R24 (t8) = 00000000
R1  (at) = 00000000  R9  (t1) = 00000000  R17 (s1) = 00000000  R25 (t9) = 00000000
R2  (v0) = 00000000  R10 (t2) = 00000000  R18 (s2) = 00000000  R26 (k0) = 00000000
R3  (v1) = 00000000  R11 (t3) = 0                                          000000

[0x00400000]    0x8fa40000  lw $4                          $sp)            # argc
[0x00400004]    0x27a50004  addiu                          $sp 4           # argv
[0x00400008]    0x24a60004  addiu                          $a1 4           # envp
[0x0040000c]    0x00041080  sll $                          a0 2
[0x00400010]    0x00c23021  addu                           $a2 $v0
[0x00400014]    0x0c100009  jal 0
[0x00400018]    0x00000000  nop

        DATA
[0x10000000]...[0x10040000]     0

        STACK
[0x7fffeffc]                    0

        KERNEL DATA

Copyright 1997 by Morgan Kaufmann Publishers, Inc.
See the file README for a full copyright notice.
Loaded: C:\Program Files\PCSpim\exceptions.s
Memory and registers cleared and the simulator reinitialized.

C:\Documents and Settings\Joshua\My Documents\CDA 3101\sum1to10.asm successfully loaded
```

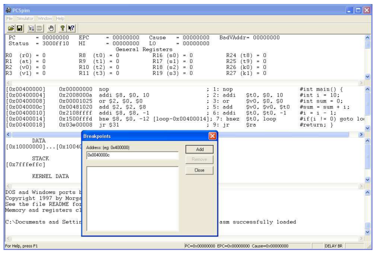
Display
☑ Save window positions
☑ General registers in hexadecimal
☐ Floating point registers in hexadecimal

Execution
☐ Bare machine ☐ Delayed Branches ☑ Mapped I/O
☑ Allow pseudo instructio ☐ Delayed Load ☐ Quiet
☑ Load exception file
Exceptio: C:\Program Files\PCSpim\exceptions.s

PC=0x00400000 EPC=0x00000000 Cause=0x00000000

- Click this menu item to change settings for the emulator.
- Sometimes it’s helpful to uncheck “General registers in hexadecimal” so that you can read values as regular numbers.

PC Spim



```
 PC      = 00000000   EPC     = 00000000   Cause    = 00000000   BadVAddr= 00000000
 Status  = 3000ff10   HI      = 00000000   LO       = 00000000
                              General Registers
R0  (r0) = 0          R8  (t0) = 0          R16 (s0) = 0          R24 (t8) = 0
R1  (at) = 0          R9  (t1) = 0          R17 (s1) = 0          R25 (t9) = 0
R2  (v0) = 0          R10 (t2) = 0          R18 (s2) = 0          R26 (k0) = 0
R3  (v1) = 0          R11 (t3) = 0          R19 (s3) = 0          R27 (k1) = 0
```

```
[0x00400000]    0x00000000  nop                              ; 1: nop                  #int main() {
[0x00400004]    0x2008000a  addi $8, $0, 10                  ; 2: addi   $t0, $0, 10   #int i = 10;
[0x00400008]    0x00001025  or $2, $0, $0                    ; 3: or     $v0, $0, $0   #int sum = 0;
[0x0040000c]    0x00481020  add $2, $2, $8                   ; 5: add    $v0, $v0, $t0 #sum = sum + i;
[0x00400010]    0x2108ffff  addi $8, $8, -1                  ; 6: addi   $t0, $t0, -1  #i = i - 1;
[0x00400014]    0x1500fffd  bne $8, $0, -12 [loop-0x00400014]; 7: bnez   $t0, loop     #if(i != 0) goto loo
[0x00400018]    0x03e00008  jr $31                           ; 9: jr     $ra           #return; }
```

```
        DATA
[0x10000000]...[0x10040

        STACK
[0x7fffeffc]

        KERNEL DATA
```

```
DOS and Windows ports b
Copyright 1997 by Morga
See the file README for
Memory and registers cl

C:\Documents and Settin                                  asm successfully loaded
```

Address: (eg: 0x400000) 0x0040000c

For Help, press F1   PC=0x00000000 EPC=0x00000000 Cause=0x00000000   DELAY BR

- Click the button that looks like a hand to set breakpoints. The program will stop running at positions you indicate, and wait for your authorization to continue upon reaching said point. You will also see the register values updated.

SPIM: Assembler, Simulator + BIOS

- Combines assembler and simulator
- Assembly language program prepared in your favourite way as a text file
- Label your first instruction as *main*, e.g.

  main: add $5, $3, $4 # comment
- Read program into SPIM which will assemble it and may indicate assembly errors (1 at a time!)
- Execute your program
- Results output to window which simulates console (or by inspection of registers)

SPIM System calls

- load argument registers
- load call code
- syscall

```
li  $a0, 10       # load argument $a0=10
li  $v0, 1        # call code to print integer
syscall           # print $a0
```

SPIM system calls

procedure	code $v0	argument
print int	1	$a0 contains number
print float	2	$f12 contains number
print double	3	$f12 contains number
print string	4	$a0 address of string

SPIM system calls

procedure	code $v0	result
read int	5	res returned in $v0
read float	6	res returned in $f0
read double	7	res returned in $f0
read string	8	

Example: Print numbers 1 to 10

```
       .data
newln:.asciiz “\n”
       .text
       .globl main
main:  li $s0, 1            # $s0 = loop counter
       li $s1, 10           # $s1 = upper bound of loop
loop:  move $a0, $s0        # print loop counter $s0
       li $v0, 1
       syscall
       li $v0, 4            # syscall for print string
       la $a0, newln        # load address of string
       syscall
       addi $s0, $s0, 1     # increase counter by 1
       ble $s0, $s1, loop   # if ($s0<=$s1) goto loop
       li $v0, 10           # exit
       syscall
```

Example: Increase array elements by 5

```
       .text
       .globl main
main:   la   $t0, Aaddr       # $t0 = pointer to array A
        lw   $t1, len         # $t1 = length (of array A)
        sll  $t1, $t1, 2      # $t1 = 4*length
        add  $t1, $t1, $t0    # $t1 = address(A)+4*length
loop:   lw   $t2, 0($t0)      # $t2 = A[i]
        addi $t2, $t2, 5      # $t2 = $t2 + 5
        sw   $t2, 0($t0)      # A[i] = $t2
        addi $t0, $t0, 4      # i = i+1
        bne  $t0, $t1, loop   # if $t0<$t1 goto loop
        .data
Aaddr:  .word 0,2,1,4,5       # array with 5 elements
len:    .word 5
```

Procedures

- jal addr
  - – store address + 4 into $ra
  - – jump to address addr
- jr $ra
  - – allows subroutine to jump back
  - – care must be taken to preserve $ra!
  - – more work for non-leaf procedures

Procedures

- one of the few means to structure your assembly language program
- small entities that can be tested separately
- can make an assembly program more readable
- recursive procedures

Write your own procedures

```
# prints the integer contained in $a0
print_int:
      li $v0, 1         # system call to
      syscall           # print integer
      jr $ra            # return

main: . . .
      li $a0, 10        # we want to print 10
      jal print_int     # print integer in $a0
```

Write your own procedures

```
      .data
newline:.asciiz “\n”
      .text
print_eol:                    # prints "\n"
      li $v0, 4               #
      la $a0, newline         #
      syscall                 #
      jr $ra                  # return
main: . . .
      jal print_eol           # printf(“\n”)
```

Write your own procedures

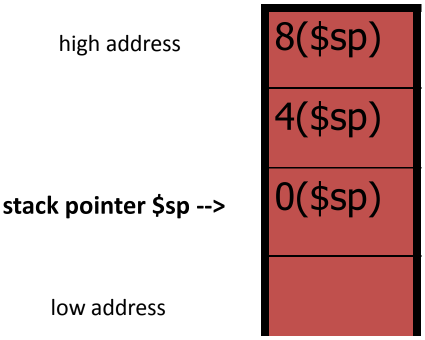
```
      .data
main:
        li $s0, 1               # $s0 = loop ctr
        li $s1, 10              # $s1 = upperbnd
loop:   move $a0, $s0           # print loop ctr
        jal print_int           #
        jal print_eol           # print "\n"
        addi $s0, $s0, 1        # loop ctr +1
        ble $s0, $s1, loop      # unless $s0>$s1…
```

Non-leaf procedures

- Suppose that a procedure procA calls another procedure jal procB
- **Problem:** jal stores return address of procedure procB and destroys return address of procedure procA
- Save $ra and all necessary variables onto the stack, call procB, and retore

The Stack

The stack can be used for
- parameter passing
- storing return addresses
- storing result variables
- stack pointer $sp


high address

stack pointer $sp -->

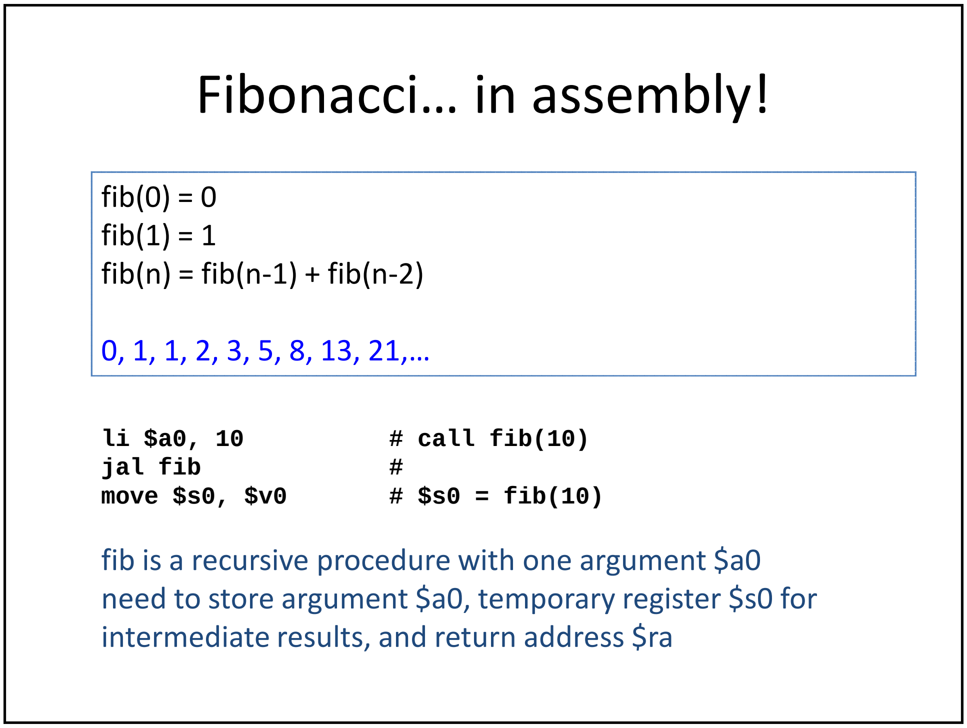
low address

8($sp)

4($sp)

0($sp)

$sp = $sp - 12



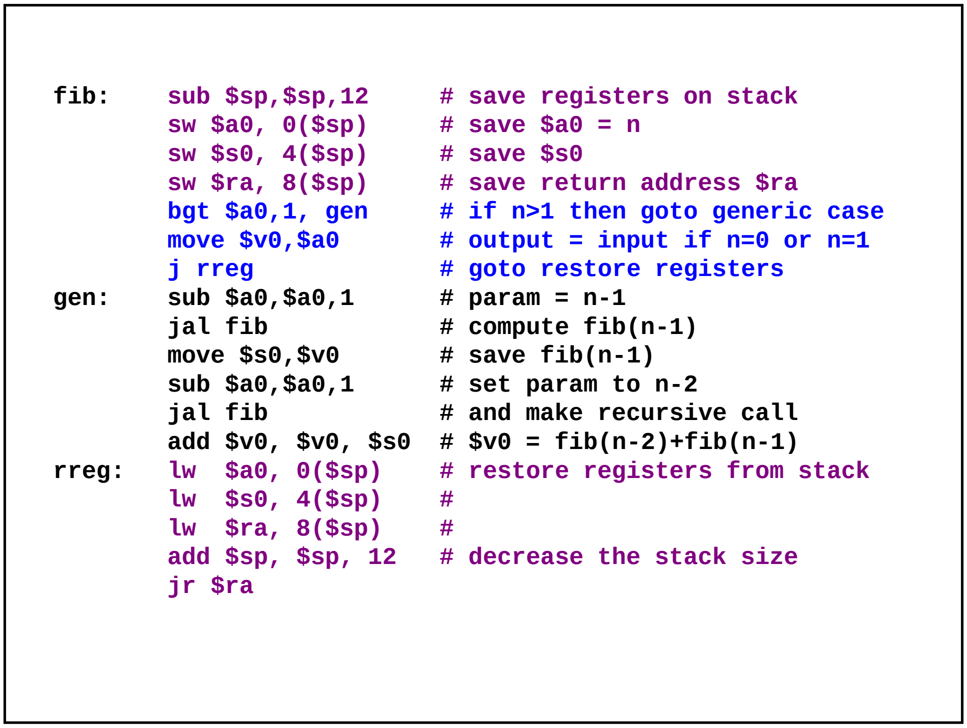
Fibonacci… in assembly!

fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2)

0, 1, 1, 2, 3, 5, 8, 13, 21,…

```
li $a0, 10          # call fib(10)
jal fib             #
move $s0, $v0       # $s0 = fib(10)
```

fib is a recursive procedure with one argument $a0
need to store argument $a0, temporary register $s0 for intermediate results, and return address $ra



```
fib:    sub $sp,$sp,12      # save registers on stack
        sw $a0, 0($sp)      # save $a0 = n
        sw $s0, 4($sp)      # save $s0
        sw $ra, 8($sp)      # save return address $ra
        bgt $a0,1, gen      # if n>1 then goto generic case
        move $v0,$a0        # output = input if n=0 or n=1
        j rreg              # goto restore registers
gen:    sub $a0,$a0,1       # param = n-1
        jal fib             # compute fib(n-1)
        move $s0,$v0        # save fib(n-1)
        sub $a0,$a0,1       # set param to n-2
        jal fib             # and make recursive call
        add $v0, $v0, $s0   # $v0 = fib(n-2)+fib(n-1)
rreg:   lw  $a0, 0($sp)     # restore registers from stack
        lw  $s0, 4($sp)     #
        lw  $ra, 8($sp)     #
        add $sp, $sp, 12    # decrease the stack size
        jr $ra
```

Optional Assembly Ticks

- Tick 0: download SPIM (some version) and assemble + run the hello world program
- Tick 1: write an assembly program which takes an array of 10 values and swaps the values (so e.g. A[0]:= A[9], A[1]:= A[8], … A[9]:= A[0])
- Tick 2: write an assembly program which reads in any 10 values from the keyboard, and prints them out lowest to highest

There will be a **prize** for the shortest correct answer to Tick 2 – email submissions to me by 21st Nov