5 MIPS Assembly Language

- Today, digital computers are almost exclusively programmed using **high-level programming languages** (PLs), *e.g.*, C, C++, Java
- The CPU fetch–execute cycle, however, is *not* prepared to directly execute high-level constructs like `if-then-else`, `do-while`, arithmetic, method invocations, *etc.*
- Instead, a CPU can execute a limited number of rather primitive instructions, its **machine language instruction set**
  - Machine language instructions are encoded as bit patterns which are interpreted during the instruction decode phase
  - A C/C++/Java **compiler** is needed to translate high-level constructs into a series of primitive machine instructions

Why machine language?

- Even with clever compilers available, machine language level programming is still of importance:
  - machine language programs can be carefully tuned for **speed** (*e.g.*, computationally heavy simulations, controlling graphics hardware)
  - the **size** of machine language programs is usually significantly smaller than the size of high-level PL code
  - specific computer features may only be available at the machine language level (*e.g.*, I/O port access in device drivers)
- For a number of small scale computers (embedded devices, wearable computers)
  - high-level PL compilers are not available yet
  - or high-level PLs are simply not adequate because compilers introduce uncertainty about the time cost of programs (*e.g.*, brake control in a car)

Machine language vs. assembly language

- *Real* machine language level programming means to handle the bit encodings of machine instructions

  Example (MIPS CPU: addition `$t0` $\leftarrow$ `$t0` + `$t1`):

  $$10000100101000000000100000$$

- **Assembly language** introduces symbolic names (**mnemonics**) for machine instructions and makes programming less error-prone:

  Example (MIPS CPU: addition `$t0` $\leftarrow$ `$t0` + `$t1`):

  ```
  add $t0, $t0, $t1
  ```

- An **assembler** translates mnemonics into machine instructions
  - Normally: mnemonic $\overset{1:1}{\longleftrightarrow}$ machine instruction
  - Also: the assembler supports **pseudo instructions** which are translated into series of machine instructions (mnemonic $\overset{1:n}{\longleftrightarrow}$ machine instruction)

The MIPS R2000/R3000 CPU

- Here we will use the MIPS CPU family to explore assembly programming
  - MIPS CPU originated from research project at Stanford, most successful and flexible CPU design of the 1990s
  - MIPS CPUs were found in SGI graphics workstations, Windows CE handhelds, CISCO routers, and Nintendo 64 video game consoles
- MIPS CPUs follow the RISC (**R**educed **I**nstruction **S**et **C**omputer) design principle:
  - limited repertoire of machine instructions
  - limited arithmetical complexity supported
  - extensive supply of CPU registers (reduce memory accesses)
- Here: work with MIPS R2000 instruction set (use MIPS R2000 simulator SPIM: `http://www.cs.wisc.edu/~larus/spim.html`)

MIPS: memory layout

- The MIPS CPU is a 32-bit architecture (all registers are 32 bits wide)
  - Accessible memory range: `0x00000000`–`0xFFFFFFFF`
- MIPS is a von-Neumann computer: memory holds both instructions (*text*) and *data*.
  - Specific memory **segments** are coventionally used to tell instructions from data:

Address	Segment
`0x7FFFFFFF`	stack
↓	↓
↑	↑
`0x10000000`	data
`0x00400000`	text
`0x00000000`	reserved

  - If a program is loaded into SPIM, its `.text` segment is automatically placed at `0x00400000`, its `.data` segment at `0x10000000`

MIPS: 32-bit, little endian

- A MIPS **word** has 32 bits (a **halfword** 16 bits, a **byte** 8 bits)
- The MIPS architecture is **little-endian**: in memory,
  a word (halfword) is stored with its *least significant byte first* 
  - **Example** (representation of 32-bit word `0x11223344` at address n):

Address	n	$n+1$	$n+2$	$n+3$
Value	`0x44`	`0x33`	`0x22`	`0x11`

  (Intel Pentium: big-endian)
- MIPS requires words (and halfwords) to be stored at **aligned addresses**:
  - if an object is of size s bytes, its storage address needs to be divisble by s (otherwise: CPU halts with *address error exception*)

MIPS: registers

- MIPS comes with 32 **general purpose registers** named `$0`...`$31`
  Registers also have symbolic names reflecting their conventional[8] use:

Register	Alias	Usage	Register	Alias	Usage
`$0`	`$zero`	constant 0	`$16`	`$s0`	saved temporary
`$1`	`$at`	used by assembler	`$17`	`$s1`	saved temporary
`$2`	`$v0`	function result	`$18`	`$s2`	saved temporary
`$3`	`$v1`	function result	`$19`	`$s3`	saved temporary
`$4`	`$a0`	argument 1	`$20`	`$s4`	saved temporary
`$5`	`$a1`	argument 2	`$21`	`$s5`	saved temporary
`$6`	`$a2`	argument 3	`$22`	`$s6`	saved temporary
`$7`	`$a3`	argument 4	`$23`	`$s7`	saved temporary
`$8`	`$t0`	unsaved temporary	`$24`	`$t8`	unsaved temporary
`$9`	`$t1`	unsaved temporary	`$25`	`$t9`	unsaved temporary
`$10`	`$t2`	unsaved temporary	`$26`	`$k0`	reserved for OS kernel
`$11`	`$t3`	unsaved temporary	`$27`	`$k1`	reserved for OS kernel
`$12`	`$t4`	unsaved temporary	`$28`	`$gp`	pointer to global data
`$13`	`$t5`	unsaved temporary	`$29`	`$sp`	stack pointer
`$14`	`$t6`	unsaved temporary	`$30`	`$fp`	frame pointer
`$15`	`$t7`	unsaved temporary	`$31`	`$ra`	return address

[8] Most of these conventions concern procedure call and return (library interoperability)

MIPS: load and store

- Typical for the RISC design, MIPS is a **load-store architecture**:
  - Memory is accessed *only* by explicit *load* and *store* instructions
  - Computation (*e.g.*, arithmetics) reads operands from registers and writes results back into registers

- MIPS: **load** word/halfword/byte at address a into target register r ($r \leftarrow (a)$):

Instruction	Remark	Pseudo?
`lw` r, a		
`lh` r, a	sign extension	
`lb` r, a	sign extension	
`lhu` r, a	no sign extension	
`lbu` r, a	no sign extension	

MIPS: load and store

- **Example** (load word/halfword/byte into temporary registers):

```
        .text
        .globl  __start
__start:
        # load with sign extension
        lw      $t0, memory
        lh      $t1, memory
        lb      $t2, memory
        # load without sign extension
        lhu     $t3, memory
        lbu     $t4, memory

        .data
memory:
        .word   0xABCDE080      # little endian: 80E0CDAB
```

Register	Value
$t0	0xABCDE080
$t1	0xFFFFE080
$t2	0xFFFFFF80
$t3	0x0000E080
$t4	0x00000080

MIPS: load and store

- MIPS: **store** word/halfword/byte in register r at address a ($a \leftarrow r$):

Instruction	Remark	Pseudo?
`sw` r, a		
`sh` r, a	stores low halfword	
`sb` r, a	stores low byte	

Example (swap values in registers `$t0` and `$t1`):

```
        .text
        .globl  __start
__start:
        # swap values $t0 and $t1 ... slow!
        sw      $t0, x
        sw      $t1, y
        lw      $t0, y
        lw      $t1, x

        .data
x:
        .word   0x000000FF
y:
        .word   0xABCDE080
```

MIPS: move

- MIPS can **move** data between registers directly (no memory access involved)

Instruction	Remark	Pseudo?
`move` r, s	target r, source s ($r \leftarrow s$)	×

Example (swap values in registers `$t0` and `$t1`, destroys `$t2`):

```
        .text
        .globl  __start
__start:
        # swap values $t0 and $t1 (clobbers $t2)
        move    $t2, $t0
        move    $t0, $t1
        move    $t1, $t2

        # no .data segment
```

– By convention, destroying the contents of the `$t`n registers is OK (`$s`n registers are assumed intact once a procedure returns )

MIPS: logical instructions

- MIPS CPUs provide instructions to compute common boolean functions

Instruction	Remark	Pseudo?
`and` r, s, t	$r \leftarrow s \cdot t$	
`andi` r, s, c	$r \leftarrow s \cdot c$ (c constant)	
`or` r, s, t	$r \leftarrow s + t$	
`ori` r, s, c	$r \leftarrow s + c$ (c constant)	
`nor` r, s, t	$r \leftarrow \overline{s + t}$	
`xor` r, s, t	$r \leftarrow s$ XOR t	
`xori` r, s, c	$r \leftarrow s$ XOR c (c constant)	
`not` r, s	$r \leftarrow \overline{s}$	$\times$

  - The `andi`, `ori`, `xori` instructions use **immediate addressing**: the constant c is encoded in the instruction bit pattern

    Example (bit pattern for instruction `andi` \$$x$, \$$y$, c with $0 \leqslant x, y \leqslant 31$):

$$\underbrace{001100}_{\texttt{andi}}\ \underbrace{bbbbb}_{y}\ \underbrace{bbbbb}_{x}\ \underbrace{bbbbbbbbbbbbbbbb}_{c\ (16\ \text{bit})}$$

MIPS: pseudo instructions

- The MIPS standard defines the CPU instruction set as well as **pseudo instructions**
- The assembler translates pseudo instructions into real MIPS instructions

  Example (translation of pseudo instructions):

Pseudo instruction	MIPS instruction	Remark
`not` r, s	`nor` r, s, `$0`	
`move` r, s	`or` r, s, `$0`	
`li` r, c	`ori` r, `$0`, c	load immediate (c: 16 bit constant)

- How does the assembler translate `li` r, `0xABCDEF00` (c in `ori` is 16 bit only)?

Pseudo instruction	MIPS instructions[9]	Remark
`li` r, `0xABCDEF00`	`lui $at, 0xABCD` `ori` r, `$at, 0xEF00`	(c: 32 bit constant)

[9] MIPS instruction: `lui` r, c: load constant halfword c into upper halfword of register r

MIPS: using pseudo instructions

- **Example** (replace the low byte of `$t0` by the low byte of `$t1`, leaving `$t0` otherwise intact—use **bitmasks** and logical instructions):

```
        .text
        .globl  __start
__start:
        li      $t0, 0x11223344
        li      $t1, 0x88776655
        # paste the low byte of $t1 into the low byte of $t0
        # ($t0 = 0x11223355)
        and     $t0, $t0, 0xFFFFFF00  # pseudo
        and     $t1, $t1, 0xFF        # assembler translates -> andi
        or      $t0, $t0, $t1

        # no .data segment
```

  - Expand the pseudo instruction:

Pseudo instruction	MIPS instructions
`and $t0, $t0, 0xFFFFFF00`	`lui $at, 0xFFFF`
	`ori $at, 0xFF00`
	`and $t0, $t0, $at`

MIPS: optimized register swap

- **Question**: swap the contents of register `$t0` and `$t1` without using memory accesses and without using temporary registers)

```
        .text
        .globl  __start
__start:
        # swap values of $t0 and $t1 (xor-based)
        xor     $t0, $t0, $t1
        xor     $t1, $t0, $t1
        xor     $t0, $t0, $t1

        # no .data segment
```

  - Explain how this "*xor swap*" works!

    Remember:

    ① a XOR $0 = a$

    ② a XOR $a = 0$

MIPS: arithmetic instructions

- MIPS provides unsigned and signed (two's complement) 32-bit integer arithmetics

Instruction	Remark	Pseudo?
`add` r, s, t	$r \leftarrow s + t$	
`addu` r, s, t	without overflow	
`addi` r, s, c	$r \leftarrow s + c$	
`addiu` r, s, c	without overflow	
`sub` r, s, t	$r \leftarrow s - t$	
`subu` r, s, t	without overflow	
`mulo` r, s, t	$r \leftarrow s \times t$	$\times$
`mul` r, s, t	without overflow	$\times$
`div` r, s, t	$r \leftarrow s/t$	$\times$
`divu` r, s, t	without overflow	$\times$

  - The 64-bit result of `mulo` (`mul`) is stored in the special registers `$hi` and `$lo`; `$lo` is moved into r[10]
  - `div` (`divu`): MIPS places the quotient in `$lo`, remainder in `$hi`; `$lo` is moved into r

[10] Access to `$lo`, `$hi`: `mflo`, `mfhi` (read) and `mtlo`, `mthi` (write)

MIPS: artihmetic and shift/rotate instructions

Instruction	Remark	Pseudo?
`abs` r, s	$r \leftarrow \lvert s \rvert$	$\times$
`neg` r, s	$r \leftarrow -s$	$\times$
`negu` r, s	without overflow	$\times$
`rem` r, s, t	$r \leftarrow$ remainder of s/t	$\times$
`remu` r, s, t	without overflow	$\times$
`sll` r, s, c	$r \leftarrow$ shift s left c bits, $r_{0\ldots c-1} \leftarrow 0$	
`sllv` r, s, t	$r \leftarrow$ shift s left t bits, $r_{0\ldots t-1} \leftarrow 0$	
`srl` r, s, c	$r \leftarrow$ shift s right c bits, $r_{31-c+1\ldots 31} \leftarrow 0$	
`srlv` r, s, t	$r \leftarrow$ shift s right t bits, $r_{31-t+1\ldots 31} \leftarrow 0$	
`sra` r, s, c	$r \leftarrow$ shift s right c bits, $r_{31-c+1\ldots 31} \leftarrow s_{31}$	
`srav` r, s, t	$r \leftarrow$ shift s right t bits, $r_{31-t+1\ldots 31} \leftarrow s_{31}$	
`rol` r, s, t	$r \leftarrow$ rotate s left t bits	$\times$
`ror` r, s, t	$r \leftarrow$ rotate s right t bits	$\times$

- **Question:** How could the assembler implement the `rol`, `ror` pseudo instructions?

MIPS: shift/rotate instructions

- MIPS assemblers implement the pseudo rotation instructions (`rol`, `ror`) based on the CPU shifting instructions:

```
        .text
        .globl  __start
__start:
        # rotate left 1 bit
        li      $t0, 0x80010004
        rol     $t1, $t0, 1

        # rotate right 3 bits
        li      $t0, 0x80010004
        ror     $t1, $t0, 3

        # no .data segment
```

→

```
        .text
        .globl  __start
__start:
        # rotate left 1 bit
        lui     $at, 0x8001
        ori     $t0, $at, 0x0004
        srl     $at, $t0, 31
        sll     $t1, $t0, 1
        or      $t1, $t1, $at

        # rotate right 3 bits
        lui     $at, 0x8001
        ori     $t0, $at, 0x004
        sll     $at, $t0, 29
        srl     $t1, $t0, 3
        or      $t1, $t1, $at

        # no .data segment
```

MIPS: branch instructions

- **Branch instructions** provide means to change the program control flow (manipulate the CPU IP register)
  - The CPU can branch unconditionally (**jump**) or depending on a specified condition (*e.g.*, equality of two registers, register $\leqslant 0$, ...)
  - In assembly programs, the branch target may be specified via a **label**—internally the branch instruction stores an 16-bit **offset**

```
        ⋮
again:  lw   $t0, memory
        sub  $t0, $t0, 1
        beqz $t0, exit      # jump offset:  2 instructions
        b    $t0, again     # jump offset:  -4 instructions
exit:   sw   $t1, memory
        ⋮
```

  - With a 16-bit offset, MIPS can branch $2^{15} - 1$ (2^{15}) instructions backward (forward)

MIPS: branch instructions

Instruction	Condition	Remark	Pseudo?	MIPS instructions
b l	none	IP $\leftarrow l$	$\times$	beq $zero, $zero, l
beq r, s, l	$r = s$			
bne r, s, l	$r \neq s$			
bgez r, l	$r \geqslant 0$			
bgtz r, l	$r > 0$			
blez r, l	$r \leqslant 0$			
bltz r, l	$r < 0$			
beqz r, l	$r = 0$		$\times$	beq $r, $zero, l
bnez r, l	$r \neq 0$		$\times$	bne $r, $zero, l
bge r, s, l	$r \geqslant s$		$\times$	slt $at, $r, s[11]
				beq $at, $zero, l
bgt r, s, l	$r > s$		$\times$	slt $at, $s, r
				bne $at, $zero, l
ble r, s, l	$r \leqslant s$		$\times$	slt $at, $s, r
				beq $at, $zero, l
blt r, s, l	$r < s$		$\times$	slt $at, $r, s
				bne $at, $zero, l

[11] slt $at, r, s: $at $\leftarrow$ 1 if $r < s$, $at $\leftarrow$ 0 otherwise.

MIPS: comparison instructions

- Compare the values of two registers (or one register and a constant)
  - Comparison $\begin{cases} \text{successful:} & r \leftarrow 1 \\ \text{fails:} & r \leftarrow 0 \end{cases}$

Instruction	Comparison	Remark	Pseudo?	MIPS instructions
`slt` r, s, t	$s < t$			
`sltu` r, s, t	$s < t$	unsigned		
`slti` r, s, c	$s < c$	c 16-bit constant		
`sltiu` r, s, c	$s < c$	unsigned		
`seq` r, s, t	$s = t$		×	`beq` $s, t, 3$ `ori` r, `$zero`, 0 `beq` `$zero`, `$zero`, 2 `ori` r, `$zero`, 1
`sne` r, s, t	$s \neq t$		×	
`sge` r, s, t	$s \geqslant t$		×	
`sgeu` r, s, t	$s \geqslant t$	unsigned	×	
`sgt` r, s, t	$s > t$		×	
`sgtu` r, s, t	$s > t$	unsigned	×	
`sle` r, s, t	$s \leqslant t$		×	
`sleu` r, s, t	$s \leqslant t$	unsigned	×	

MIPS: division by zero, overflow

- Arithmetic exceptions (division by 0, overflow during multiplication) are handled on the MIPS instruction layer itself:
  - MIPS instructions for `div $t0, $t1, $t2`:

```
bne   $t2, $zero, 2
break $0              # exception: divsion by 0
div   $t1, $t2        # quotient in $lo, remainder in $hi
mflo  $t0
```

  - MIPS instructions for `mulo $t0, $t1, $t2`:

```
mult  $t1, $t2        # result bits 31..0 in $lo, 63..32 in $hi
mfhi  $at
mflo  $t0
sra   $t0, $t0, 31    # $t0 = 0x00000000 or 0xFFFFFFFF (sign bit)
beq   $at, $t0, 2
break $0              # exception: arithmetic overflow
mflo  $t0
```

Compute Fibonacci numbers (iteratively)

- The **Fibonacci numbers**[12] are an infinite sequence of positive integers (originally used to describe the development of rabbit poulations)
  - Start of sequence:

    $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, \ldots$
  - The n-th Fibonacci number is **recursively defined** as follows:

$$fib(n) = \begin{cases} 0 & \text{if n} = 0 \\ 1 & \text{if n} = 1 \\ fib(n-2) + fib(n-1) & \text{if } n \geqslant 2 \end{cases}$$

Example (compute $fib(3)$):

$$fib(3) = fib(1) + fib(2) = 1 + fib(0) + fib(1) = 1 + 0 + 1 = 2$$

[12] Leonardo Fibonacci (Leonardo di Pisa), 1200

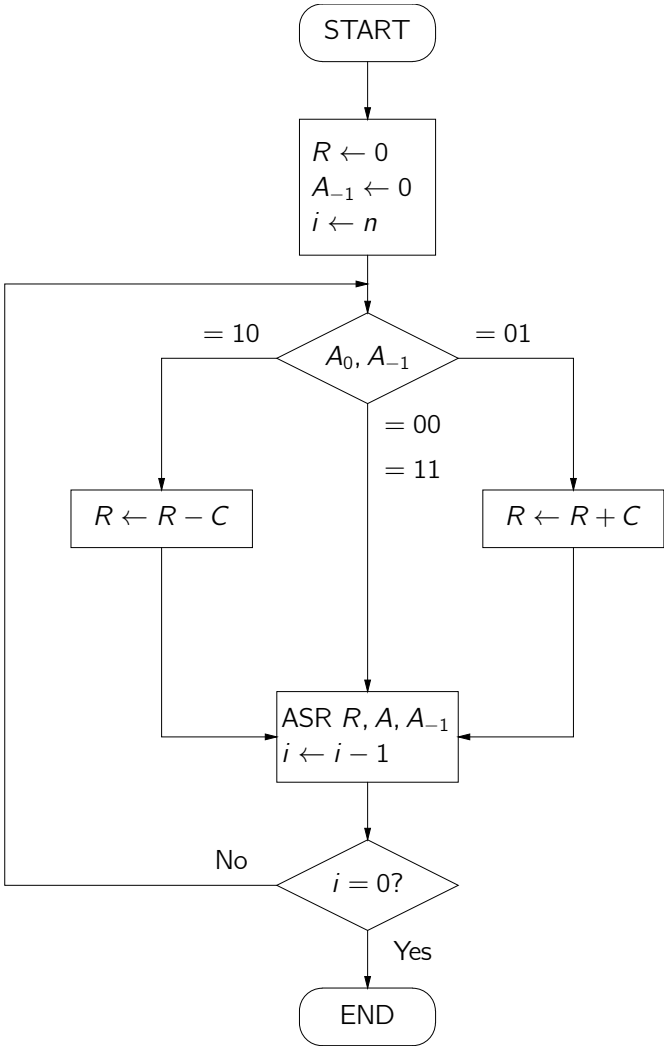
Compute Fibonacci numbers (iteratively)

Register	Usage
$a0	parameter *n*
$v0	last Fibonacci number computed so far (and result)
$t0	second last Fibonacci number computed so far
$t1	temporary scratch register

```
        .text
        .globl __start
__start:
        li      $a0, 1                  # fib(n): parameter n

        move    $v0, $a0                # n < 2 => fib(n) = n
        blt     $a0, 2, done
        li      $t0, 0                  # second last Fib' number
        li      $v0, 1                  # last Fib' number
fib:    add     $t1, $t0, $v0           # compute next Fib' number in sequence
        move    $t0, $v0                # update second last
        move    $v0, $t1                # update last
        sub     $a0, $a0, 1             # more work to do?
        bgt     $a0, 1, fib             # yes: iterate again
done:   sw      $v0, result             # no:  store result, done

        .data
result: .word   0x11111111
```

MIPS: Booth's algorithm

- Remember Booth's algorithm to multiply two's complement numbers
  - Note: equivalent functionality is provided by MIPS instruction `mult`



  - Register assignment:

Register	Usage
`$a0`	A
`$a1`	C
`$v0`	R
`$t0`	i
`$t1`	A_{-1} (only bit 0 of `$t1` used)

  - Implementation quite straightforward, ASR of "connected registers" R, A, A_{-1} needs some care

MIPS: Booth's algorithm

```
        .text
        .globl  __start
__start:
        li      $a0, -5                 # parameter A
        li      $a1, 7                  # parameter C

        li      $v0, 0                  # R <- 0
        li      $t1, 0                  # A(-1) <- 0
        li      $t0, 32                 # i <- n (32 bits)
booth:
        and     $t2, $a0, 0x00000001    # $t2 <- A0
        sll     $t2, $t2, 1
        or      $t2, $t2, $t1           # $t2 = A0,A(-1)

        beq     $t2, 2, case10          # $t2 = 10?
        beq     $t2, 1, case01          # $t2 = 01?
        b       shift                   # $t2 = 00 or $t2 = 11
case10: sub     $v0, $v0, $a1           # R <- R - C
        b       shift
case01: add     $v0, $v0, $a1           # R <- R + C
shift:
        and     $t1, $a0, 0x00000001    # A(-1) <- A0
        and     $t2, $v0, 0x00000001    # save R0
        sll     $t2, $t2, 31
        srl     $a0, $a0, 1             # shift right A
        or      $a0, $a0, $t2           # A31 <- R0
        sra     $v0, $v0, 1             # arithmetic shift right R

        sub     $t0,$t0, 1              # i <- i - 1
        bnez    $t0, booth              # i = 0?
                                        # result in $v0,$a0
```

MIPS: Addressing modes

- A MIPS instruction like

  `lb $t0`, *memory*

  addresses a given, *fixed* address in memory.

- Commonly, however, programs need to **access consecutive ranges of memory addresses** (*e.g.*, to perform **string processing**)

  Example (place string representation in the data segment):

```
        .data
str:    .asciiz "foobar"    # null-terminated ASCII encoded string
```

  Content of data segment at address `str`:

Address	`str+0`	`str+1`	`str+2`	`str+3`	`str+4`	`str+5`	`str+6`
Value	102	111	111	98	97	114	0

MIPS: Addressing modes

- **Assembler instructions** available to place constant sequences of data into data segment:

Assembler instruction	Data
`.ascii` s	ASCII encoded characters of string s
`.asciiz` s	like `.ascii`, null-terminated
`.word` $w_1, w_2, \ldots$	32-bit words $w_1, w_2, \ldots$
`.half` $h_1, h_2, \ldots$	16-bit halfwords $h_1, h_2, \ldots$
`.byte` $b_1, b_2, \ldots$	8-bit bytes $b_1, b_2, \ldots$
`.float` $f_1, f_2, \ldots$	32-bit single precision floating point numbers $f_1, f_2, \ldots$
`.double` $d_1, d_2, \ldots$	64-bit double precision floating point numbers $d_1, d_2, \ldots$
`.space` n	n zero bytes

- To consecutively access all characters (bytes, halfwords, words) in such a sequence, the CPU needs to **compute the next address** to access

MIPS: Indirect addressing

- **Indirect Addressing**: address is held in a register

  Example:

```
        .text

        la      $t0, str
        lb      $t1, ($t0)    # access byte at address $t0 ('f')
        add     $t0, $t0, 3
        lb      $t2, ($t0)    # access byte at address $t0 + 3 ('b')

        .data

str:    .asciiz "foobar"
```

-  Note the difference between

  `lw $t0,` *a* and `la $t0,` *a*

MIPS: Indexed addressing

- Actually, in keeping with the RISC philosophy, MIPS has only one general memory addressing mode: **indexed addressing**
  - In indexed addressing, addresses are of the form (16-bit constant c, CPU register r)

    $$c(r)$$

    - r holds a 32-bit address to which the signed 16-bit constant c is *added* to form the final address

    Example (repeated from last slide):

```
        .text

        la      $t0, str
        lb      $t1, 0($t0)    # access byte at address $t0 ('f')
        lb      $t2, 3($t0)    # access byte at address $t0 + 3 ('b')

        .data

str:    .asciiz "foobar"
```

MIPS: Indexed addressing

- **Example** (copy a sequence of n bytes from address `src` to address `dst`):

```
        .text
        .globl  __start
__start:
        # length n of byte sequence - 1
        li      $t0, 5
copy:
        lb      $t1, src($t0)   # pseudo! (src: 32 bits wide)
        sb      $t1, dst($t0)
        sub     $t0, $t0, 1
        bgez    $t0, copy

        .data

src:    .byte 0x11, 0x22, 0x33, 0x44, 0x55, 0x66
dst:    .space 6
```

- **Questions:** which changes are necessary to turn this into a n word (n halfword) copy routine?

MIPS: Optimized copy routine

- Copying byte sequences via `lb`/`sb` is inefficient on von-Neumann machines

```
        .text
        .globl  __start
__start:
        li      $a0, 11             # length n of byte sequence
        la      $a1, src            # source address
        la      $a2, dst            # destination address

        and     $t1, $a0, 0x03
        srl     $t0, $a0, 2
copy:   beqz    $t0, rest
        lw      $t2, ($a1)
        sw      $t2, ($a2)
        add     $a1, $a1, 4
        add     $a2, $a2, 4
        sub     $t0, $t0, 1
        b       copy
rest:   beqz    $t1, done
        lb      $t2, ($a1)
        sb      $t2, ($a2)
        add     $a1, $a1, 1
        add     $a2, $a2, 1
        sub     $t1, $t1, 1
        b       rest
done:
        .data

        .align 4
src:    .byte 0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0x99, 0xAA
        .align 4
dst:    .space 11
```

MIPS: Addressing modes (summary)

Mode	Example	MIPS instruction(s)	Remark [Address]
immediate	`andi $t0, $t0, 0x03`		16-bit constant embedded in instruction
IP relative	`beqz $t0, done`		signed 16-bit jump offset o embedded in instruction [IP $+ 4 \times o$]
direct	`lw $t0, 0x11223344`	`lui $at, 0x1122` `lw $t0, 0x3344($at)`	[0x11223344]
indirect	`lw $t0, ($t1)`	`lw $t0, 0($t1)`	[$t1]

indexed	`lw $t0, 0x11223344($t1)`	`lui $at, 0x1122`
		`addu $at, $at, $t1`
		`lw $t0, 0x3344($at)`
			[0x11223344 + $t1]

SPIM: System calls

- The MIPS emulator SPIM provides a few services (**system calls**) which would normally be provided by the underlying **operating system**
  - Most importantly, these services provide basic console input/output (I/O) functionality to read/write numbers and strings

Service	Call code	Arguments	Result
print integer	1	`$a0`: integer	
print null-term. string	4	`$a0`: string address	
read integer	5		`$v0`: integer
read string	8	`$a0`: buffer address, `$a1`: length	
exit	10		

Remarks:

- Place system call code in `$v0`, then execute `syscall`
- System call *read integer* reads an entire line (including newline) and ignores characters following the number
- The *read string* system call reads at most `$a1` − 1 characters into the buffer and terminates the input with a null byte

SPIM: System calls

- **Example** (read integer n from SPIM console, then print $42 \times n$ on console):

```
        .text
        .globl  __start
__start:
        li      $v0, 5                  # read integer
        syscall
        mul     $t0, $v0, 42            # compute and save result

        li      $v0, 4                  # print string
        la      $a0, the_result_is
        syscall

        li      $v0, 1                  # print integer
        move    $a0, $t0
        syscall

        li      $v0, 10                 # exit
        syscall

        .data

the_result_is:
        .asciiz "The result is "
```

SPIM: System calls

- **Example** (read integer *n*, then print hexadecimal equivalent on console):
  - **Idea**: groups of 4 bits correspond to one hexadecimal digit (`0`...`F`), use value of group (0...15) as index into table of hexadecimal digits

```
        .text
        .globl  __start
__start:
        li      $v0, 5                  # read integer n
        syscall
        move    $t0, $v0                # save n

        li      $t1, 7                  # 7 (+ 1) hex digits for 32 bits
hexify: and     $t2, $t0, 0x0F          # extract least significant 4 bits
        srl     $t0, $t0, 4             # prepare for next digit
        lb      $t3, hex_table($t2)     # convert 4 bit group to hex digit
        sb      $t3, hex_digits($t1)    # store hex digit
        sub     $t1, $t1, 1             # next digit
        bgez    $t1, hexify             # more digits?

        li      $v0, 4                  # print string
        la      $a0, the_result_is
        syscall
        li      $v0, 10                 # exit
        syscall

        .data

hex_table:      .ascii  "0123456789ABCDEF"
the_result_is:  .ascii  "Hexadecimal value: 0x"
hex_digits:     .asciiz "XXXXXXXX"
```

Bubble sort

- **Bubble sort** is a simple sorting algorithm (with *quadratic complexity*: to sort n items, bubble sort in general needs n^2 steps to complete)
  - **Input:** *array* A of n items (numbers, strings, ...) and an ordering $<$, *e.g.*, $n = 5$:

$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$
3	4	10	5	3

  - **Output:** sorted array A, *e.g.*:

$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$
3	3	4	5	10

- **Basic idea:**

① $j \leftarrow n - 1$ (index of last element in A)
② If $A[j] < A[j-1]$, swap both elements
③ $j \leftarrow j - 1$, goto ② if $j > 0$
④ Goto ① if a swap occurred

Bubble sort (trace)

① $i = 4$

$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$
3	4	10	5	③

② $j = 3$

$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$
3	4	10	③ ↔	5

③ $A[3] = 3$

$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$
3	4	10	③	5

② $j = 2$

$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$
3	4	③ ↔	10	5

③ $A[2] = 3$

$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$
3	4	③	10	5

② $j = 1$

$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$
3	③ ↔	4	10	5

③ $A[1] = 3$

$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$
3	③	4	10	5

② $j = 0$

$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$
③ ↔	3	4	10	5

④ Swap occured? (Yes, goto ①)

① $i = 3$

$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$
3	3	4	10	⑤

MIPS: Bubble sort

```
        .text
        .globl  __start
__start:
        li      $a0, 10                 # parameter n

        sll     $a0, $a0, 2             # number of bytes in array A
outer:
        sub     $t0, $a0, 8             # $t0: j-1
        li      $t1, 0                  # no swap yet
inner:
        lw      $t2, A+4($t0)           # $t2 <- A[j]
        lw      $t3, A($t0)             # $t3 <- A[j-1]
        bgt     $t2, $t3, no_swap       # A[j] <= A[j-1]?

        sw      $t2, A($t0)             # A[j-1] <- $t2  \ move bubble
        sw      $t3, A+4($t0)           # A[j] <- $t3    / $t2 upwards
        li      $t1, 1                  # swap occurred
no_swap:
        sub     $t0, $t0, 4             # next array element
        bgez    $t0, inner              # more?

        bnez    $t1, outer              # did we swap?

        li      $v0, 10                 # exit
        syscall

        .data

A:                                      # array A (sorted in-place)
        .word   4,5,6,7,8,9,10,2,1,3
```

Procedures (sub-routines)

- The solution to a complex programming problem is almost always assembled from simple program pieces (**procedures**) which constitute a small building block of a larger solution

- Often, procedures provide some service which can then be requested by the **main program** in *many places*

  – Instead of copying and repeating the procedure code over and over,
    ① the main program **calls** the procedure
    (jumps to the procedure code),
    ② the called procedure does its job before it **returns** control
    (jumps back to the instruction *just after* the procedure call)
  – The main program is often referred to as the **caller**, the procedure as the **callee**

MIPS: Procedure example

- **Example** (procedure to compute the average of two parameters x, y):
  - Input: parameter x in `$a0`, parameter y in `$a1`
  - Output: average of x, y in `$v0` $\left(\texttt{\$v0} \leftarrow \frac{\texttt{\$a0} + \texttt{\$a1}}{2}\right)$

```
        .text

        # procedure average (x,y)
        #   input:  x in $a0, y in $a1
        #   output: $v0
average:
        add     $v0, $a0, $a1     # $v0 <- $a0 + $a1
        sra     $v0, $v0, 1       # $v0 <- $v0 / 2

        j       ???               #  where to return to?
```

MIPS: Procedure call

- A typical main program (caller of `average`) might look like as follows:

```
            .text

            :
            li      $a0, $t0            # set parameter x
            li      $a1, 12             # set parameter y
            j       average             # compute average
★1:         move    $t0, $v0            # save result
            :

            :
            li      $a0, $t0            # set parameter x
            li      $a1, $t1            # set parameter y
            j       average             # compute average
★2:         move    $t1, $v0            # save result
            :
```

- After the first call, `average` needs to return to label ★1, after the second call the correct address to return to is ★2

MIPS: Procedure call and return

- MIPS instruction `jal` (*jump and link*) jumps to the given address *a* (procedure entry point) and records the correct **return address** in register `$ra`:

Instruction	Effect
`jal` *a*	`$ra` $\leftarrow$ IP $+ 4$ IP $\leftarrow$ *a*

- The callee may then simply return to the correct address in the caller via

Instruction	Effect
`j $ra`	IP $\leftarrow$ `$ra`

  - `$ra` is reserved MIPS CPU register `$31`; programs overwriting/abusing `$ra` are likely to yield chaos

MIPS: Procedure call and return[13]

```
            .text

            :
            li      $a0, $t0            # set parameter x
            li      $a1, 12             # set parameter y
            jal     average             # compute average ($ra = ★1)
★1:         move    $t0, $v0            # save result
            :
            li      $a0, $t0            # set parameter x
            li      $a1, $t1            # set parameter y
            jal     average             # compute average ($ra = ★2)
★2:         move    $t1, $v0            # save result
            :
            li      $v0, 10             # exit program
            syscall

            # procedure average (x,y)
            #   input:  x in $a0, y in $a1
            #   output: $v0
average:
            add     $v0, $a0, $a1     # $v0 <- $a0 + $a1
            sra     $v0, $v0, 1       # $v0 <- $v0 / 2
            j       $ra               # return to caller
```

[13] NB: The ★ labels merely illustrate the effect of `jal` and do not appear in the real assembly file

Recursive algorithms/procedures

- Some algorithms solve a complex problem as follows:

  ① Is the size of the problem such that we can trivially solve it?
  Yes ⇒ Return the answer immediately

  ② Try to reduce the size/complexity of the original problem

  ③ **Invoke the algorithm** on the reduced problem

- In step ③, the algorithm *invokes itself* to compute the answer; such algorithms are known as **recursive**

- Recursion may also be used in MIPS assembly procedures, typically:

```
        .text
        ⋮
proc:   ...          # code for procedure proc
        ⋮
        jal proc     # recursive call
        ⋮
```

Binary search

- **Binary search** is a recursive algorithm that searches for a given value (*needle*) in a *sorted array* of values

- General idea:

  ① If the array is of size 1 only, compare *needle* against the array entry, return result of comparison (*true*/*false*)
  ② Locate the middle array entry m; compare *needle* and m
  ③ – If $needle = m$ return *true*
  – If $needle < m$, **call binary search** on left half of array
  – Otherwise, **call binary search** on right half of array

- NB: since the array is sorted, we know for all entries a to the left of m that $a \leqslant m$ (for all entries b to the right of m: $b \geqslant m$)

Binary search (example, needle = 35)

①

$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$
2	3	8	10	16	21	35	42	43	50

②

$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$
2	3	8	10	16	21	35	42	43	50

③

					$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$
2	3	8	10	16	21	35	42	43	50

②

					$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$
2	3	8	10	16	21	35	42	43	50

③

					$A[5]$	$A[6]$	$A[7]$		
2	3	8	10	16	21	35	42	43	50

②

					$A[5]$	$A[6]$	$A[7]$		
2	3	8	10	16	21	35	42	43	50

③

					$A[5]$	$A[6]$	$A[7]$		
2	3	8	10	16	21	**35**	42	43	50

Binary search: efficiency

- Binary search is very efficient: in each recursive call, the size of the problem is cut in half
  - Let the original array come with n entries; each recursive call halves the problem size:

$$n \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdots$$

  - Binary search is guaranteed to end after s recursive calls when the array size has been reduced to 1:

$$n \cdot \left(\frac{1}{2}\right)^s \stackrel{!}{=} 1 \quad \Leftrightarrow \quad \frac{n}{2^s} = 1 \quad \Leftrightarrow \quad s = \log_2 n$$

array size n	$\lceil \log_2 n \rceil$
10	4
100	7
1000	10
10000	14
100000	17
1000000	20

MIPS: Recursive procedures

- Note that the MIPS procedure call/return via `jal` *a*/`j $ra` does *not* work for recursive procedures

- **What is going wrong?**
  Using `jal` in callee *overwrites* register `$ra` which is later needed to return to caller:

```
        .text
        ⋮
        # main program
        ⋮
        jal proc          # invoke procedure ($ra = ★1)
★1:     ⋮

        # procedure
proc:   ⋮
        jal proc          # recursive call ($ra = ★2)
★2:     ⋮

        j   $ra           #  (will jump to ★2, not ★1)
```

MIPS: Save/restore `$ra` in recursive procedure

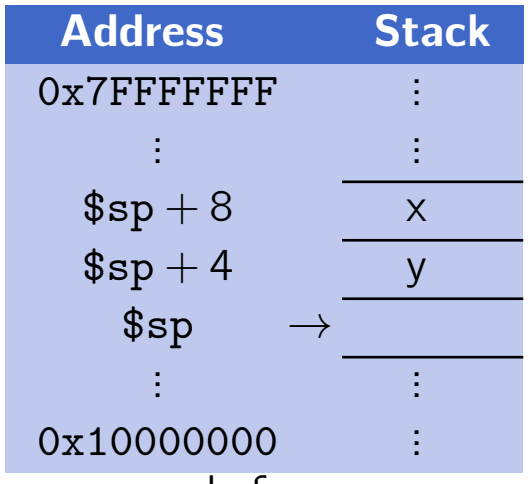
```
        .text
        ⋮
        # main program
        ⋮
        jal proc              # invoke procedure ($ra = ★1)
★1:     ⋮

        # procedure
proc:   save $ra
        ⋮
        jal proc              # recursive call ($ra = ★2)
★2:     ⋮
        restore $ra
        j   $ra               # will jump to ★1
```

NB: **Each invocation of `proc` needs its own place to save `$ra`**—the save location will otherwise be overwritten in the recursive call!

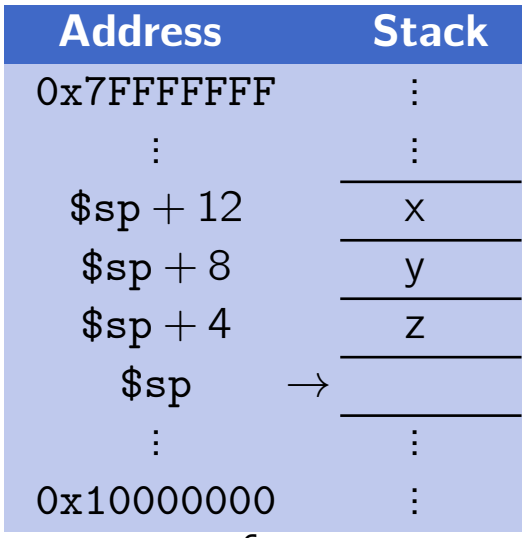
MIPS: Saving registers on the stack

- Conventionally, procedures **save registers on the stack** if they need to preserve register values
  - A **stack** is an area of memory that may grow/shrink depending on the actual space needed by a program
  - CPU register `$sp` points *just below* the last object stored on the stack
  - **Pushing** more items on the stack moves `$sp` and lets the stack grow
  - **Example** (MIPS: push 32-bit word z on stack):

```
subu $sp, $sp, 4
sw   z, 4($sp)
```



before



after

MIPS: Saving registers on the stack

```
1           .text
2           .globl  __start
3   __start:
4           li      $a0, 1          # (0)
5           jal     proc
6   done:
7           li      $v0, 10
8           syscall
9
10  proc:
11          subu    $sp, $sp, 4
12          sw      $ra, 4($sp)     # (1),(2)
13
14          beqz    $a0, return
15          li      $a0, 0
16          jal     proc
17
18  return:
19          lw      $ra, 4($sp)
20          addu    $sp, $sp, 4     # (3),(4)
21          j       $ra
22
23          # no .data segment
```

State of stack at time (t):

(0)

Address	Stack
0x7FFFFFFF	⋮
⋮	⋮
$sp →	???
⋮	⋮
0x10000000	⋮

(1)

Address	Stack
0x7FFFFFFF	⋮
⋮	⋮
$sp + 4	done
$sp →	???
⋮	⋮
0x10000000	⋮

MIPS: Saving registers on the stack

```
1           .text
2           .globl  __start
3   __start:
4           li      $a0, 1          # (0)
5           jal     proc
6   done:
7           li      $v0, 10
8           syscall
9
10  proc:
11          subu    $sp, $sp, 4
12          sw      $ra, 4($sp)     # (1),(2)
13
14          beqz    $a0, return
15          li      $a0, 0
16          jal     proc
17
18  return:
19          lw      $ra, 4($sp)
20          addu    $sp, $sp, 4     # (3),(4)
21          j       $ra
22
23          # no .data segment
```

State of stack at time (t):

(2)

Address		Stack
0x7FFFFFFF		⋮
⋮		⋮
$sp + 8		done
$sp + 4		return
$sp	→	???
⋮		⋮
0x10000000		⋮

(3)

Address		Stack
0x7FFFFFFF		⋮
⋮		⋮
$sp + 4		done
$sp	→	return
		???
⋮		⋮
0x10000000		⋮

MIPS: Saving registers on the stack

```
1          .text
2          .globl  __start
3  __start:
4          li      $a0, 1          # (0)
5          jal     proc
6  done:
7          li      $v0, 10
8          syscall
9
10 proc:
11         subu    $sp, $sp, 4
12         sw      $ra, 4($sp)     # (1),(2)
13
14         beqz    $a0, return
15         li      $a0, 0
16         jal     proc
17
18 return:
19         lw      $ra, 4($sp)
20         addu    $sp, $sp, 4     # (3),(4)
21         j       $ra
22
23         # no .data segment
```

State of stack at time ⓣ:

④

Address		Stack
0x7FFFFFFF		⋮
⋮		⋮
$sp	→	done
		return
		???
⋮		⋮
0x10000000		⋮

- NB: the memory pointed to by $sp and below is considered garbage

MIPS: Recursive binary search

- **Example** (recursive binary search procedure, `$ra` saved on stack):

```
1         .data
2
3  first:  # sorted array of 32 bit words
4         .word   2, 3, 8, 10, 16, 21, 35, 42, 43, 50, 64, 69
5         .word   70, 77, 82, 83, 84, 90, 96, 99, 100, 105, 111, 120
6  last:   # address just after sorted array
7
8         .text
9         .globl  __start
10 __start:
11
12        # binary search in sorted array
13        #   input:  search value (needle) in $a0
14        #           base address of array in $a1
15        #           last address of array in $a2
16        #   output: address of needle in $v0 if found,
17        #           0 in $v0 otherwise
18        li      $a0, 42                 # needle value
19        la      $a1, first              # address of first array entry
20        la      $a2, last - 4           # address of last array entry
21        jal     binsearch               # perform binary search
22
23        li      $v0, 10
24        syscall
25
```

MIPS: Recursive binary search (cont.)

```
26 binsearch:
27         subu    $sp, $sp, 4             # allocate 4 bytes on stack
28         sw      $ra, 4($sp)             # save return address on stack
29
30         subu    $t0, $a2, $a1           # $t0 <- size of array
31         bnez    $t0, search             # if size > 0, continue search
32
33         move    $v0, $a1                # address of only entry in array
34         lw      $t0, ($v0)              # load the entry
35         beq     $a0, $t0, return        # equal to needle value? yes => return
36         li      $v0, 0                  # no => needle not in array
37         b       return                  # done, return
38 search:
39         sra     $t0, $t0, 3             # compute offset of middle entry m:
40         sll     $t0, $t0, 2             #   $t0 <- ($t0 / 8) * 4
41         addu    $v0, $a1, $t0           # compute address of middle entry m
42         lw      $t0, ($v0)              # $t0 <- middle entry m
43         beq     $a0, $t0, return        # m = needle? yes => return
44         blt     $a0, $t0, go_left       # needle less than m? yes =>
45                                         # search continues left of m
46 go_right:
47         addu    $a1, $v0, 4             # search continues right of m
48         jal     binsearch               # recursive call
49         b       return                  # done, return
50 go_left:
51         move    $a2, $v0                # search continues left of m
52         jal     binsearch               # recursive call
53 return:
54         lw      $ra, 4($sp)             # recover return address from stack
55         addu    $sp, $sp, 4             # release 4 bytes on stack
56
57         j       $ra                     # return to caller
```