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 $t0 \leftarrow t0 + t1$):

  \[
  100001001010000000100000
  \]

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

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

  \[
  \text{add } t0, t0, t1
  \]

- An **assembler** translates mnemonics into machine instructions
  - Normally: mnemonic $\stackrel{1:1}{\rightarrow}$ machine instruction
  - Also: the assembler supports **pseudo instructions** which are translated into series of machine instructions (mnemonic $\stackrel{1:n}{\rightarrow}$ 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 (Reduced Instruction Set Computer) design principle:
  – limited repertoire of machine instructions
  – limited arithmetical complexity supported
  – extensive supply of CPU registers (reduce memory accesses)

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 conventionally used to tell instructions from data:

<table>
<thead>
<tr>
<th>Address</th>
<th>Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7FFFFFFF</td>
<td>stack</td>
</tr>
<tr>
<td>↓</td>
<td>↓</td>
</tr>
<tr>
<td>↑</td>
<td>↑</td>
</tr>
<tr>
<td>0x10000000</td>
<td>data</td>
</tr>
<tr>
<td>0x00400000</td>
<td>text</td>
</tr>
<tr>
<td>0x00000000</td>
<td>reserved</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Address</th>
<th>n</th>
<th>n + 1</th>
<th>n + 2</th>
<th>n + 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>0x44</td>
<td>0x33</td>
<td>0x22</td>
<td>0x11</td>
</tr>
</tbody>
</table>

(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 divisible 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 use:

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

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 \( r \) (\( r \leftarrow (a) \)):

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Remark</th>
<th>Pseudo?</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw ( r, a )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lh ( r, a )</td>
<td>sign extension</td>
<td></td>
</tr>
<tr>
<td>lb ( r, a )</td>
<td>sign extension</td>
<td></td>
</tr>
<tr>
<td>lhu ( r, a )</td>
<td>no sign extension</td>
<td></td>
</tr>
<tr>
<td>lbu ( r, a )</td>
<td>no sign extension</td>
<td></td>
</tr>
</tbody>
</table>
MIPS: load and store

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

```assembly
.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
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t0</td>
<td>0xABCDE080</td>
</tr>
<tr>
<td>$t1</td>
<td>0xFFFFFE080</td>
</tr>
<tr>
<td>$t2</td>
<td>0xFFFFFFFF80</td>
</tr>
<tr>
<td>$t3</td>
<td>0x0000E080</td>
</tr>
<tr>
<td>$t4</td>
<td>0x00000080</td>
</tr>
</tbody>
</table>
MIPS: load and store

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Remark</th>
<th>Pseudo?</th>
</tr>
</thead>
<tbody>
<tr>
<td>sw $r, a</td>
<td>stores low halfword</td>
<td></td>
</tr>
<tr>
<td>sh $r, a</td>
<td>stores low byte</td>
<td></td>
</tr>
</tbody>
</table>

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

```assembly
.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)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Remark</th>
<th>Pseudo?</th>
</tr>
</thead>
<tbody>
<tr>
<td>move ( r, s )</td>
<td>target ( r ), source ( s ) ((r \leftarrow s))</td>
<td>×</td>
</tr>
</tbody>
</table>

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

```assembly
.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 \( \$tn \) registers is OK (\( \$sn \) registers are assumed intact once a procedure returns 🍬)
MIPS CPUs provide instructions to compute common boolean functions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Remark</th>
<th>Pseudo?</th>
</tr>
</thead>
<tbody>
<tr>
<td>and r, s, t</td>
<td>( r \leftarrow s \cdot t )</td>
<td></td>
</tr>
<tr>
<td>andi r, s, c</td>
<td>( r \leftarrow s \cdot c ) (c constant)</td>
<td></td>
</tr>
<tr>
<td>or r, s, t</td>
<td>( r \leftarrow s + t )</td>
<td></td>
</tr>
<tr>
<td>ori r, s, c</td>
<td>( r \leftarrow s + c ) (c constant)</td>
<td></td>
</tr>
<tr>
<td>nor r, s, t</td>
<td>( r \leftarrow \overline{s + t} )</td>
<td></td>
</tr>
<tr>
<td>xor r, s, t</td>
<td>( r \leftarrow s \text{ XOR } t )</td>
<td></td>
</tr>
<tr>
<td>xori r, s, c</td>
<td>( r \leftarrow s \text{ XOR } c ) (c constant)</td>
<td></td>
</tr>
<tr>
<td>not r, s</td>
<td>( r \leftarrow \overline{s} )</td>
<td>( \times )</td>
</tr>
</tbody>
</table>

- 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 \(0 \leq x, y \leq 31\)):
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):

<table>
<thead>
<tr>
<th>Pseudo instruction</th>
<th>MIPS instruction</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>not r, s</td>
<td>nor r, s, $0</td>
<td></td>
</tr>
<tr>
<td>move r, s</td>
<td>or r, s, $0</td>
<td></td>
</tr>
<tr>
<td>li r, c</td>
<td>ori r, $0, c</td>
<td>load immediate (c: 16 bit constant)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Pseudo instruction</th>
<th>MIPS instructions⁹</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>li r, 0xABCDEF00</td>
<td>lui $at, 0xABCD</td>
<td>(c: 32 bit constant)</td>
</tr>
<tr>
<td></td>
<td>ori r, $at, 0xEF00</td>
<td></td>
</tr>
</tbody>
</table>

⁹MIPS 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):

```assembly
.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:

<table>
<thead>
<tr>
<th>Pseudo instruction</th>
<th>MIPS instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>and $t0,$t0,0xFFFFFFFF00</td>
<td>lui $at,0xFFFF</td>
</tr>
<tr>
<td></td>
<td>ori $at,0xFF00</td>
</tr>
<tr>
<td></td>
<td>and $t0,$t0,$at</td>
</tr>
</tbody>
</table>
MIPS: optimized register swap

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

```plaintext
.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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Remark</th>
<th>Pseudo?</th>
</tr>
</thead>
<tbody>
<tr>
<td>add r, s, t</td>
<td>$r \leftarrow s + t$</td>
<td></td>
</tr>
<tr>
<td>addu r, s, t</td>
<td>without overflow</td>
<td></td>
</tr>
<tr>
<td>addi r, s, c</td>
<td>$r \leftarrow s + c$</td>
<td></td>
</tr>
<tr>
<td>addiu r, s, c</td>
<td>without overflow</td>
<td></td>
</tr>
<tr>
<td>sub r, s, t</td>
<td>$r \leftarrow s - t$</td>
<td></td>
</tr>
<tr>
<td>subu r, s, t</td>
<td>without overflow</td>
<td></td>
</tr>
<tr>
<td>mulo r, s, t</td>
<td>$r \leftarrow s \times t$</td>
<td>$\times$</td>
</tr>
<tr>
<td>mul r, s, t</td>
<td>without overflow</td>
<td>$\times$</td>
</tr>
<tr>
<td>div r, s, t</td>
<td>$r \leftarrow s/t$</td>
<td>$\times$</td>
</tr>
<tr>
<td>divu r, s, t</td>
<td>without overflow</td>
<td>$\times$</td>
</tr>
</tbody>
</table>

- 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: arithmetic and shift/rotate instructions

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

**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
```
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 ≤ 0, ...)
- In assembly programs, the branch target may be specified via a label—internally the branch instruction stores an 16-bit offset

```assembly
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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Remark</th>
<th>Pseudo?</th>
<th>MIPS instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>b</code> /</td>
<td>none</td>
<td>IP ← /</td>
<td>×</td>
<td><code>beq $zero,$zero, /</code></td>
</tr>
<tr>
<td><code>beq r, s, /</code></td>
<td><code>r = s</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>bne r, s, /</code></td>
<td><code>r ≠ s</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>bgez r, /</code></td>
<td><code>r ≥ 0</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>bgtz r, /</code></td>
<td><code>r &gt; 0</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>blez r, /</code></td>
<td><code>r ≤ 0</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>bltz r, /</code></td>
<td><code>r &lt; 0</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>beqz r, /</code></td>
<td><code>r = 0</code></td>
<td>×</td>
<td></td>
<td><code>beq r,$zero, /</code></td>
</tr>
<tr>
<td><code>bnez r, /</code></td>
<td><code>r ≠ 0</code></td>
<td>×</td>
<td></td>
<td><code>bne r,$zero, /</code></td>
</tr>
<tr>
<td><code>bge r, s, /</code></td>
<td><code>r ≥ s</code></td>
<td>×</td>
<td></td>
<td><code>slt $at,r,s$</code></td>
</tr>
<tr>
<td><code>bgt r, s, /</code></td>
<td><code>r &gt; s</code></td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>ble r, s, /</code></td>
<td><code>r ≤ s</code></td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>blt r, s, /</code></td>
<td><code>r &lt; s</code></td>
<td>×</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

11 $slt \$at,r,s$: $\$at ← 1$ if $r < s$, $\$at ← 0$ otherwise.
MIPS: comparison instructions

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comparison</th>
<th>Remark</th>
<th>Pseudo?</th>
<th>MIPS instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>slt r, s, t</td>
<td>(s &lt; t)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sltu r, s, t</td>
<td>(s &lt; t)</td>
<td>unsigned</td>
<td></td>
<td></td>
</tr>
<tr>
<td>slti r, s, c</td>
<td>(s &lt; c)</td>
<td>c 16-bit constant</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sltiu r, s, c</td>
<td>(s &lt; c)</td>
<td>unsigned</td>
<td></td>
<td></td>
</tr>
<tr>
<td>seq r, s, t</td>
<td>(s = t)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sne r, s, t</td>
<td>(s \neq t)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sge r, s, t</td>
<td>(s \geq t)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sgeu r, s, t</td>
<td>(s \geq t)</td>
<td>unsigned</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sgt r, s, t</td>
<td>(s &gt; t)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sgtu r, s, t</td>
<td>(s &gt; t)</td>
<td>unsigned</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sle r, s, t</td>
<td>(s \leq t)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sleu r, s, t</td>
<td>(s \leq t)</td>
<td>unsigned</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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: division 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 populations)
  
  - 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:

\[
\begin{align*}
  \text{fib}(n) &= \begin{cases}
    0 & \text{if } n = 0 \\
    1 & \text{if } n = 1 \\
    \text{fib}(n-2) + \text{fib}(n-1) & \text{if } n \geq 2
  \end{cases}
\end{align*}
\]

**Example** (compute \(\text{fib}(3)\)):

\[
\text{fib}(3) = \text{fib}(1) + \text{fib}(2) = 1 + \text{fib}(0) + \text{fib}(1) = 1 + 0 + 1 = 2
\]

\(^{12}\)Leonardo Fibonacci (Leonardo di Pisa), 1200
Compute Fibonacci numbers (iteratively)

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a0</td>
<td>parameter $n</td>
</tr>
<tr>
<td>$v0</td>
<td>last Fibonacci number computed so far (and result)</td>
</tr>
<tr>
<td>$t0</td>
<td>second last Fibonacci number computed so far</td>
</tr>
<tr>
<td>$t1</td>
<td>temporary scratch register</td>
</tr>
</tbody>
</table>

```
.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:

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a0</td>
<td>A</td>
</tr>
<tr>
<td>$a1</td>
<td>C</td>
</tr>
<tr>
<td>$v0</td>
<td>R</td>
</tr>
<tr>
<td>$t0</td>
<td>i</td>
</tr>
<tr>
<td>$t1</td>
<td>$A_{-1}$ (only bit 0 of $t1$ used)</td>
</tr>
</tbody>
</table>

- 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
  \[
  \text{lb } \$t0, \text{ memory}
  \]
  addresses a given, \textit{fixed} address in memory.

- Commonly, however, programs need to \textbf{access consecutive ranges of memory addresses} (\textit{e.g.}, to perform \textbf{string processing})

\textbf{Example} (place string representation in the data segment):

```
.data

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

Content of data segment at address \texttt{str}:

<table>
<thead>
<tr>
<th>Address</th>
<th>str+0</th>
<th>str+1</th>
<th>str+2</th>
<th>str+3</th>
<th>str+4</th>
<th>str+5</th>
<th>str+6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>102</td>
<td>111</td>
<td>111</td>
<td>98</td>
<td>97</td>
<td>114</td>
<td>0</td>
</tr>
</tbody>
</table>
**MIPS: Addressing modes**

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

<table>
<thead>
<tr>
<th>Assembler instruction</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>.ascii s</td>
<td>ASCII encoded characters of string s</td>
</tr>
<tr>
<td>.asciiz s</td>
<td>like .ascii, null-terminated</td>
</tr>
<tr>
<td>.word w₁, w₂,...</td>
<td>32-bit words w₁, w₂,...</td>
</tr>
<tr>
<td>.half h₁, h₂,...</td>
<td>16-bit halfwords h₁, h₂,...</td>
</tr>
<tr>
<td>.byte b₁, b₂,...</td>
<td>8-bit bytes b₁, b₂,...</td>
</tr>
<tr>
<td>.float f₁, f₂,...</td>
<td>32-bit single precision floating point numbers f₁, f₂,...</td>
</tr>
<tr>
<td>.double d₁, d₂,...</td>
<td>64-bit double precision floating point numbers d₁, d₂,...</td>
</tr>
<tr>
<td>.space n</td>
<td>n zero bytes</td>
</tr>
</tbody>
</table>

- 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:

```assembly
.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.

```assembly
.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)

<table>
<thead>
<tr>
<th>Mode</th>
<th>Example</th>
<th>MIPS instruction(s)</th>
<th>Remark [Address]</th>
</tr>
</thead>
<tbody>
<tr>
<td>immediate</td>
<td><code>andi $t0, $t0, 0x03</code></td>
<td></td>
<td>16-bit constant embedded in instruction</td>
</tr>
<tr>
<td>IP relative</td>
<td><code>beqz $t0, done</code></td>
<td></td>
<td>signed 16-bit jump offset ( o ) embedded in instruction [IP + 4 \times o]</td>
</tr>
<tr>
<td>direct</td>
<td><code>lw $t0, 0x11223344</code></td>
<td><code>lui $at, 0x1122</code> <code>lw $t0, 0x3344($at)</code></td>
<td>[0x11223344]</td>
</tr>
<tr>
<td>indirect</td>
<td><code>lw $t0, ($t1)</code></td>
<td><code>lw $t0, 0($t1)</code></td>
<td>[$t1]</td>
</tr>
<tr>
<td>indexed</td>
<td><code>lw $t0, 0x11223344($t1)</code></td>
<td><code>lui $at, 0x1122</code> <code>addu $at, $at, $t1</code> <code>lw $t0, 0x3344($at)</code></td>
<td>[0x11223344 + $t1]</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Service</th>
<th>Call code</th>
<th>Arguments</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>print integer</td>
<td>1</td>
<td>$a0: integer</td>
<td>$v0: integer</td>
</tr>
<tr>
<td>print null-term. string</td>
<td>4</td>
<td>$a0: string address</td>
<td></td>
</tr>
<tr>
<td>read integer</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read string</td>
<td>8</td>
<td>$a0: buffer address, $a1: length</td>
<td></td>
</tr>
<tr>
<td>exit</td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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):

```asm
.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

```assembly
.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., \( n = 5 \):

  \[
  3 \quad 4 \quad 10 \quad 5 \quad 3
  \]

- **Output:** sorted array \( A \), e.g.:

  \[
  3 \quad 3 \quad 4 \quad 5 \quad 10
  \]

- **Basic idea:**

  1. \( j \leftarrow n - 1 \) (index of last element in \( A \))
  3. \( j \leftarrow j - 1 \), goto 2 if \( j > 0 \)
  4. Goto 1 if a swap occurred
Bubble sort (trace)

   \[3 \ 4 \ 10 \ 5\]

   \[3 \ 4 \ 10 \ 5\]

   \[3 \ 4 \ 10 \ 5\]

   \[3 \ 4 \ 10 \ 5\]

   \[3 \ 4 \ 10 \ 5\]

   \[3 \ 4 \ 10 \ 5\]

   \[3 \ 4 \ 10 \ 5\]

   \[3 \ 4 \ 10 \ 5\]

   \[3 \ 4 \ 10 \ 5\]

    \[3 \ 4 \ 10 \ 5\]

Swap occurred? (Yes, goto 1)
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:
    .word 4,5,6,7,8,9,10,2,1,3 # array A (sorted in-place)
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
• **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 $v0$ ($v0 \leftarrow \frac{a0 + a1}{2}$)

  .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:

```assembly
.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 `$ra$:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>jal <code>a</code></td>
<td><code>$ra ← IP + 4$</code></td>
</tr>
<tr>
<td></td>
<td><code>IP ← a</code></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>j $ra</code></td>
<td><code>IP ← $ra</code></td>
</tr>
</tbody>
</table>

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

.text

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

13NB: 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 $\Rightarrow$ 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:

  ```assembly
  .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* \(\leq m\) (for all entries *b* to the right of *m*: *b* \(\geq m\)).
Binary search (example, needle = 35)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>①</td>
<td>2</td>
<td>3</td>
<td>8</td>
<td>10</td>
<td>16</td>
<td>21</td>
<td>35</td>
<td>42</td>
<td>43</td>
<td>50</td>
</tr>
<tr>
<td>②</td>
<td>2</td>
<td>3</td>
<td>8</td>
<td>10</td>
<td>16</td>
<td>21</td>
<td>35</td>
<td>42</td>
<td>43</td>
<td>50</td>
</tr>
<tr>
<td>③</td>
<td>2</td>
<td>3</td>
<td>8</td>
<td>10</td>
<td>16</td>
<td>21</td>
<td>35</td>
<td>42</td>
<td>43</td>
<td>50</td>
</tr>
</tbody>
</table>
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:
    $$n \cdot \left(\frac{1}{2}\right)^s = 1 \iff \frac{n}{2^s} = 1 \iff s = \log_2 n$$

<table>
<thead>
<tr>
<th>array size $n$</th>
<th>$\lceil \log_2 n \rceil$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>10</td>
</tr>
<tr>
<td>10000</td>
<td>14</td>
</tr>
<tr>
<td>100000</td>
<td>17</td>
</tr>
<tr>
<td>1000000</td>
<td>20</td>
</tr>
</tbody>
</table>
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

```assembly
.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)
    ```

    ![Stack Diagram](image)

    **before**

    ![Stack Diagram](image)

    **after**
MIPS: Saving registers on the stack

```
.text
.globl __start
__start:
li   $a0, 1   # (0)
jal  proc
done:
li   $v0, 10
syscall

proc:
subu $sp, $sp, 4
sw   $ra, 4($sp)   # (1),(2)
beqz $a0, return
li   $a0, 0
jal  proc
return:
lw   $ra, 4($sp)
addu $sp, $sp, 4   # (3),(4)
j   $ra

# no .data segment
```

State of stack at time ①:

<table>
<thead>
<tr>
<th>Address</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7FFFFFFF</td>
<td>:</td>
</tr>
<tr>
<td>$sp</td>
<td>→</td>
</tr>
<tr>
<td>0x10000000</td>
<td>:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7FFFFFFF</td>
<td>:</td>
</tr>
<tr>
<td>$sp + 4</td>
<td>done</td>
</tr>
<tr>
<td>$sp</td>
<td>→</td>
</tr>
<tr>
<td>0x10000000</td>
<td>:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7FFFFFFF</td>
<td>:</td>
</tr>
<tr>
<td>$sp</td>
<td>→</td>
</tr>
<tr>
<td>0x10000000</td>
<td>:</td>
</tr>
</tbody>
</table>
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. beqz $a0, return
14. li $a0, 0
15. jal proc
16. return:
17. lw $ra, 4($sp)
18. addu $sp, $sp, 4  # (3),(4)
19. j $ra
20. # no .data segment

State of stack at time 🕒:

<table>
<thead>
<tr>
<th>Address</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7FFFFFFFFF</td>
<td></td>
</tr>
<tr>
<td>$sp + 8</td>
<td>done</td>
</tr>
<tr>
<td>$sp + 4</td>
<td>return</td>
</tr>
<tr>
<td>$sp</td>
<td>→</td>
</tr>
<tr>
<td>0x10000000</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7FFFFFFFFF</td>
<td></td>
</tr>
<tr>
<td>$sp + 4</td>
<td>done</td>
</tr>
<tr>
<td>$sp</td>
<td>→</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0x10000000</td>
<td></td>
</tr>
</tbody>
</table>
MIPS: Saving registers on the stack

```
.text
.globl __start
__start:
    li   $a0, 1  # (0)
    jal  proc
done:
    li    $v0, 10
    syscall
proc:
    subu $sp, $sp, 4
    sw    $ra, 4($sp)  # (1),(2)
    beqz $a0, return
    li    $a0, 0
    jal   proc
return:
    lw    $ra, 4($sp)
    addu $sp, $sp, 4  # (3),(4)
    j     $ra
    # no .data segment
```

State of stack at time (1):

<table>
<thead>
<tr>
<th>Address</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7FFFFFFF</td>
<td>:</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>$sp</td>
<td>→ done</td>
</tr>
<tr>
<td>return</td>
<td></td>
</tr>
<tr>
<td>0x10000000</td>
<td>:</td>
</tr>
</tbody>
</table>

• 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):

```mips
.data

first:  # sorted array of 32 bit words
.word  2, 3, 8, 10, 16, 21, 35, 42, 43, 50, 64, 69
.word  70, 77, 82, 83, 84, 90, 96, 99, 100, 105, 111, 120
last:   # address just after sorted array

.text
.globl __start
__start:

  # binary search in sorted array
  # input:  search value (needle) in $a0
  #        base address of array in $a1
  #        last address of array in $a2
  # output: address of needle in $v0 if found,
  #        0 in $v0 otherwise
  li   $a0, 42                   # needle value
  la   $a1, first               # address of first array entry
  la   $a2, last - 4            # address of last array entry
  jal  binsearch               # perform binary search

  li   $v0, 10
  syscall
```
MIPS: Recursive binary search (cont.)

binsearch:
```
26  subu     $sp, $sp, 4                      # allocate 4 bytes on stack
27  sw       $ra, 4($sp)                     # save return address on stack
28  subu     $t0, $a2, $a1                 # $t0 <- size of array
29  bnez     $t0, search                  # if size > 0, continue search
30  move     $v0, $a1                      # address of only entry in array
31  lw       $t0, ($v0)                     # load the entry
32  beq      $a0, $t0, return            # equal to needle value? yes => return
33  li        $v0, 0                      # no => needle not in array
34  b        return                       # done, return

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  go_right:
```
```
46  addu     $a1, $v0, 4                   # search continues right of m
47  jal      binsearch                   # recursive call
48  b        return                      # done, return

go_left:
```
```
51  move     $a2, $v0                     # search continues left of m
52  jal      binsearch                   # recursive call
53  return:
```
```
57  lw       $ra, 4($sp)                  # recover return address from stack
58  addu     $sp, $sp, 4                  # release 4 bytes on stack
59  j        $ra                          # return to caller