Why Study Assembler?

Useful to know assembly language because ...

• sometimes you are *required* to use it  (e.g. device handlers)
• improves your understanding of how compiled programs execute
  • very helpful when debugging
  • understand performance issues better
• performance tweaking    (squeezing out last pico-s)
  • re-write that performance critical code in assembler
A typical modern CPU has

- a set of data registers
- a set of control registers (incl PC)
- an arithmetic-logic unit (ALU)
- access to memory (RAM)
- a set of simple instructions
  - transfer data between memory and registers
  - push values through the ALU to compute results
  - make tests and transfer control of execution

Different types of processors have different configurations of the above
CPU Architecture

![CPU Architecture Diagram]
Fetch-Execute Cycle

All CPUs have program execution logic like:

```c
uint32_t pc = STARTING_ADDRESS;
while (1) {
    uint32_t instruction = memory[pc];
    pc++;  // move to next instr
    if (instruction == HALT) {
        break;
    } else {
        execute(instruction);
    }
}
```

pc = Program Counter, a CPU register which tracks execution
Note that some instructions may modify pc (e.g. JUMP)
Fetch-Execute Cycle

Executing an instruction involves

- determine what the *operator* is
- determine which *registers*, if any, are involved
- determine which *memory location*, if any, is involved
- carry out the operation with the relevant operands
- store result, if any, in appropriate register

Example instruction encodings (not from a real machine):

![Instruction Diagram](image)
Assembly Language

Instructions are simply bit patterns within a 32-bit bit-string. Could specify machine code as a sequence of hex digits, e.g.

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100000</td>
<td>0x3c041001</td>
</tr>
<tr>
<td>0x100004</td>
<td>0x34020004</td>
</tr>
<tr>
<td>0x100008</td>
<td>0x0000000c</td>
</tr>
<tr>
<td>0x10000C</td>
<td>0x03e00008</td>
</tr>
</tbody>
</table>

Assembly language is a symbolic way of specifying machine code:

- write instructions using mnemonics rather than hex codes
- refer to registers using either numbers or names
- can associate names to memory addresses
MIPS is a well-known and simple architecture

- historically used everywhere from supercomputers to PlayStations, ...
- still popular in some embedded fields e.g. modems, TVs
- but being out-competed by ARM (in phones, ...)

We consider the MIPS32 version of the MIPS family

- qtspim ... provides a GUI front-end, useful for debugging
- spim ... command-line based version, useful for testing
- xspim ... GUI front-end, useful for debugging, only in CSE labs

Executables and source: http://spimsimulator.sourceforge.net/
Source code for browsing under /home/cs1521/spim
MIPS Instructions

MIPS has several classes of instructions:

- *load and store* .. transfer data between registers and memory
- *computational* ... perform arithmetic/logical operations
- *jump and branch* ... transfer control of program execution
- *coprocessor* ... standard interface to various co-processors
- *special* ... miscellaneous tasks (e.g. syscall)

And several *addressing modes* for each instruction

- between memory and register (direct, indirect)
- constant to register (immediate)
- register + register + destination register
MIPS instructions are 32-bits long, and specify ...

- an operation (e.g. load, store, add, branch, ...)
- one or more operands (e.g. registers, memory addresses, constants)

Some possible instruction formats
Examples MIPS Assembler

lw   $t1,address  # reg[t1] = memory[address]
sw   $t3,address  # memory[address] = reg[t3]
        # address must be 4-byte aligned
la   $t1,address  # reg[t1] = address
lui  $t2,const   # reg[t2] = const <<< 16
and  $t0,$t1,$t2  # reg[t0] = reg[t1] & reg[t2]
add  $t0,$t1,$t2  # reg[t0] = reg[t1] + reg[t2]
        # add signed 2’s complement ints
addi $t2,$t3, 5  # reg[t2] = reg[t3] + 5
        # add immediate, no sub immediate
mult $t3,$t4    # (Hi,Lo) = reg[t3] * reg[t4]
        # store 64-bit result in
        # registers Hi,Lo
seq  $t7,$t1,$t2  # reg[t7] = (reg[t1] == reg[t2])
j    label       # PC = label
beq  $t1,$t2,label # PC = label if reg[t1]==reg[t2]
nop            # do nothing
MIPS CPU has

- 32 general purpose registers (32-bit)
- 16/32 floating-point registers (for float/double)
- PC ... 32-bit register (always aligned on 4-byte boundary)
- HI,LO ... for storing results of multiplication and division

Registers can be referred to as $0..$31 or by symbolic names
Some registers have special uses e.g.

- register $0$ always has value 0, cannot be written
- registers $1$, $26$, $27$ reserved for use by system

More details on following slides ...
### MIPS Architecture - Integer Registers

<table>
<thead>
<tr>
<th>Number</th>
<th>Names</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$zero</td>
<td>Constant 0</td>
</tr>
<tr>
<td>1</td>
<td>$at</td>
<td>Reserved for assembler</td>
</tr>
<tr>
<td>2,3</td>
<td>$v0,$v1</td>
<td>Expression evaluation and results of a function</td>
</tr>
<tr>
<td>4..7</td>
<td>$a0..$a3</td>
<td>Arguments 1-4</td>
</tr>
<tr>
<td>8..16</td>
<td>$t0..$t7</td>
<td>Temporary (not preserved across function calls)</td>
</tr>
<tr>
<td>16..23</td>
<td>$s0..$s7</td>
<td>Saved temporary (preserved across function calls)</td>
</tr>
<tr>
<td>24,25</td>
<td>$t8,$t9</td>
<td>Temporary (preserved across function calls)</td>
</tr>
<tr>
<td>26,27</td>
<td>$k0,$k1</td>
<td>Reserved for OS kernel</td>
</tr>
<tr>
<td>28</td>
<td>$gp</td>
<td>Pointer to global area</td>
</tr>
<tr>
<td>29</td>
<td>$sp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>30</td>
<td>$fp</td>
<td>Frame pointer</td>
</tr>
<tr>
<td>31</td>
<td>$ra</td>
<td>Return address (used by function call instruction)</td>
</tr>
</tbody>
</table>

- Except for registers 0 and 31, these uses are only conventions.
- Conventions allow compiled code from different sources to be combined (linked).
- Most of these conventions are irrelevant when you are writing small MIPS assembly code programs.
- But use registers 8..23 for holding values.
- Definitely do not use 0,1 and 31.
# MIPS Architecture - floating point registers

<table>
<thead>
<tr>
<th>Reg</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f0..f2</td>
<td>hold return value of functions which return floating-point results</td>
</tr>
<tr>
<td>$f4..f10</td>
<td>temporary registers; not preserved across function calls</td>
</tr>
<tr>
<td>$f12..f14</td>
<td>used for first two double-precision function arguments</td>
</tr>
<tr>
<td>$f16..f18</td>
<td>temporary registers; used for expression evaluation</td>
</tr>
<tr>
<td>$f20..f30</td>
<td>saved registers; value is preserved across function calls</td>
</tr>
</tbody>
</table>

## Notes:

- floating point registers can be used as 32 32-bit register or 16 64-bit registers
- for 64-bit use even numbered registers
Data and Addresses

All operations refer to data, either

- in a register
- in memory
- literally (i.e. constant)

Computation operations refer to registers or constants. Only load/store instructions refer to memory. To access registers, you can also use $name$
e.g. $zero == 0$, $t0 == 8$, $fp == 30$, ...
To refer to literals, use C-like constants:

```
1 3 -1 -2 12345 0x1 0xFFFFFFFF
"a string" 'a' 'b' '1' '\n' '\0'
```
## Memory Addressing Modes

Ways of specifying memory addresses:

<table>
<thead>
<tr>
<th>Format</th>
<th>Address referred to</th>
</tr>
</thead>
<tbody>
<tr>
<td>(reg)</td>
<td>contents of register</td>
</tr>
<tr>
<td></td>
<td>e.g. ($8) ($fp) ($5)</td>
</tr>
<tr>
<td>imm</td>
<td>immediate (= constant)</td>
</tr>
<tr>
<td></td>
<td>e.g. 0 0x80000000 123456789</td>
</tr>
<tr>
<td>imm(reg)</td>
<td>immediate + contents of register</td>
</tr>
<tr>
<td></td>
<td>e.g. 0($fp) 0x80000000($9)</td>
</tr>
<tr>
<td>sym</td>
<td>address of symbol (= name)</td>
</tr>
<tr>
<td></td>
<td>e.g. main endloop exit</td>
</tr>
<tr>
<td>sym(reg)</td>
<td>address of symbol + reg contents</td>
</tr>
<tr>
<td></td>
<td>e.g. main($fp) array($9)</td>
</tr>
<tr>
<td>sym +/- imm</td>
<td>address of symbol +/- immediate</td>
</tr>
<tr>
<td></td>
<td>e.g. main+8 endloop-4</td>
</tr>
<tr>
<td>sym +/- imm(reg)</td>
<td>sym address +/- ( imm + reg contents)</td>
</tr>
<tr>
<td></td>
<td>e.g. main+8($8) array+0($18)</td>
</tr>
</tbody>
</table>
Describing MIPS Assembler Operations

• An *address* refers to a memory cell.
• \( \text{Mem}[addr] = \) contents of cell at address \( addr \).
• \&\text{name} = location of memory cell for \text{name}.

• Registers are denoted:
  \( R_d \) destination register (where result goes)
  \( R_s \) source register \#1 (where data comes from)
  \( R_t \) source register \#2 (where data comes from)

• \( \text{Reg}[R] = \) contents of register \( R \).

• Data transfer is denoted by \(<-\).

\begin{verbatim}
lw $7, buffer # Reg[7] <- Mem[buffer]
\end{verbatim}
Setting Register

\[
\begin{align*}
\text{li } & \quad R_d \text{ imm} & \text{load immediate} & \quad \text{Reg}[R_d] \leftarrow \text{imm} \\
\text{la } & \quad R_d \text{ addr} & \text{load address} & \quad \text{Reg}[R_d] \leftarrow \text{addr} \\
\text{move } & \quad R_d \text{ } R_s & \text{move data reg-to-reg} & \quad \text{Reg}[R_d] \leftarrow \text{Reg}[R_s]
\end{align*}
\]
Setting Register

- The *li* (load immediate) instruction is used to set a register to a constant value, e.g.

  ```
  li $8, 42   # $8 = 42
  li $24, 0x2a # $24 = 42
  li $15, '*'  # $15 = 42
  ```

- The *move* instruction is used to set a register to the same value as another register, e.g.

  ```
  move $8, $9 # assign to $8 value in $9
  ```

- Note destination is first register.
Setting A Register to An Address

- Note the *la* (load address) instruction is used to set a register to a labelled memory address.

```
la $8, start
```

- The memory address will be fixed before the program is run, so this differs only syntactically from the *li* instruction.
- For example, if *vec* is the label for memory address 0x10000100 then these two instructions are equivalent:

```
la $7, vec
li $7, 0x10000100
```

- In both cases the constant is encoded as part of the instruction.
- Neither *la* or *li* access memory - there are very different to the *lw* instruction.
• Both \textit{la} and \textit{li} are pseudo instructions provided by the assembler for user convenience.
• The assembler translates these pseudo-instructions into instructions actually implemented by the processor.
• For example, \texttt{li $7, 15} \ might be translated to \texttt{addi $7, $0, 15}
• If the constant is large the assembler will need to translate a \texttt{li/la} instruction to two actual instructions.
Accessing Memory

These instructions move data between memory and CPU. 1, 2 and 4-bytes (8, 16 and 32 bit) quantities can be moved. There are two operands the register which will supply/receive the value and the memory address. For the 1 and 2-byte operations the low (least significant) bits of the register are used.

\[
\begin{align*}
\text{lw } R_d & addr \quad \text{load word (32-bits)} \\
& \quad \text{Reg}[R_d] \leftarrow \text{Mem}[\text{addr}..\text{addr+3}] \\
\text{sw } R_s & addr \quad \text{store word (32-bits)} \\
& \quad \text{Mem}[\text{addr}..\text{addr+3}] \leftarrow \text{Reg}[R_s] \\
\text{lh } R_d & addr \quad \text{load half-word (16 bits)} \\
& \quad \text{Reg}[R_d] \leftarrow \text{Mem}[\text{addr}..\text{addr+1}] \\
\text{sh } R_d & addr \quad \text{store half-word (16 bits)} \\
& \quad \text{Mem}[\text{addr}..\text{addr+1}] \leftarrow \text{Reg}[R_s] \\
\text{lb } R_d & addr \quad \text{load byte (8-bits)} \\
& \quad \text{Reg}[R_d] \leftarrow \text{Mem}[\text{addr}] \\
\text{sb } R_d & addr \quad \text{store byte (8-bits)} \\
& \quad \text{Mem}[\text{addr}] \leftarrow \text{Reg}[R_s]
\end{align*}
\]
Addressing Modes

Examples of load/store and addressing:

```assembly
main:
    la $t0, vec        # reg[t0] = &vec[0]
    li $t1, 5          # reg[t1] = 5
    sw $t1, ($t0)      # vec[0] = reg[t1]
    li $t1, 13         # reg[t1] = 13
    sw $t1, 4($t0)     # vec[1] = reg[t1]
    li $t1, -7         # reg[t1] = -7
    sw $t1, 8($t0)     # vec[2] = reg[t1]
    li $t2, 12         # reg[t2] = 12
    li $t1, 42         # reg[t1] = 42
    sw $t1, vec($t2)   # vec[3] = reg[t1]
    jr $ra             # return
.data
vec: .space 16       # int vec[4];
```
MIPS instructions can manipulate different-sized operands
  • single bytes, two bytes ("halfword"), four bytes ("word")
Many instructions also have variants for signed and unsigned
Leads to many opcodes for a (conceptually) single operation, e.g.
  • LB ... load one byte from specified address
  • LBU ... load unsigned byte from specified address
  • LH ... load two bytes from specified address
  • LHU ... load unsigned 2-bytes from specified address
  • LW ... load four bytes (one word) from specified address
  • LA ... load the specified address
All of the above specify a destination register
Arithmetic Instructions

\[
\text{add } R_d R_s R_t \quad \text{addition} \\
\text{Reg}[R_d] \leftarrow \text{Reg}[R_s] + \text{Reg}[R_t]
\]

\[
\text{add } R_d R_s \text{imm} \quad \text{addition} \\
\text{Reg}[R_d] \leftarrow \text{Reg}[R_s] + \text{imm}
\]

\[
\text{sub } R_d R_s R_t \quad \text{subtraction} \\
\text{Reg}[R_d] \leftarrow \text{Reg}[R_s] - \text{Reg}[R_t]
\]

\[
\text{mul } R_d R_s R_t \quad \text{multiplication} \\
\text{Reg}[R_d] \leftarrow \text{Reg}[R_s] \times \text{Reg}[R_t]
\]

\[
\text{div } R_d R_s R_t \quad \text{division} \\
\text{Reg}[R_d] \leftarrow \text{Reg}[R_s] / \text{Reg}[R_t]
\]

\[
\text{rem } R_d R_s R_t \quad \text{remainder} \\
\text{Reg}[R_d] \leftarrow \text{Reg}[R_s] \% \text{Reg}[R_t]
\]

\[
\text{neg } R_d R_s \quad \text{negate} \\
\text{Reg}[R_d] \leftarrow - \text{Reg}[R_s]
\]

All arithmetic is signed (2’s-complement).
The second operand \((R_t)\) can be replaced by a constant in all the above instructions.
Unsigned versions of instructions are available
e.g. addu, subu, mulu, divu, ...
Logic Instructions

and \( R_dR_sR_t \)  \hspace{1cm} \text{logical} \hspace{0.5cm} \text{and}  \\
\hspace{3cm} \text{Reg}[R_d] \leftarrow \text{Reg}[R_s] \ & \text{Reg}[R_t]  \\
and \( R_dR_s\text{imm} \)  \hspace{1cm} \text{logical} \hspace{0.5cm} \text{and}  \\
\hspace{3cm} \text{Reg}[R_d] \leftarrow \text{Reg}[R_s] \ & \text{imm}  \\
or \( R_dR_sR_t \)  \hspace{1cm} \text{logical} \hspace{0.5cm} \text{or}  \\
\hspace{3cm} \text{Reg}[R_d] \leftarrow \text{Reg}[R_s] \ | \text{Reg}[R_t]  \\
\text{not} \ R_dR_s \hspace{1cm} \text{logical} \hspace{0.5cm} \text{not}  \\
\hspace{3cm} \text{Reg}[R_d] \leftarrow \text{Reg}[R_s]  \\
\text{xor} \ R_dR_sR_t \hspace{1cm} \text{logical} \hspace{0.5cm} \text{xor}  \\
\hspace{3cm} \text{Reg}[R_d] \leftarrow \text{Reg}[R_s] \ \hat{\text{Reg}}[R_t]  \\

All of these instructions can use \text{imm} instead of \( R_t \).
Bit Manipulation Instructions

\begin{align*}
\text{sll} & \quad R_d R_s R_t & \text{shift left logical} & \quad \text{Reg}[R_d] \leftarrow \text{Reg}[R_s] \ll \text{Reg}[R_t] \\
\text{sll} & \quad R_d R_s \text{imm} & \text{shift left logical} & \quad \text{Reg}[R_d] \leftarrow \text{Reg}[R_s] \ll \text{imm} \\
\text{srl} & \quad R_d R_s \text{imm} & \text{shift right logical} & \quad \text{Reg}[R_d] \leftarrow \text{Reg}[R_s] \gg \text{imm} \\
\text{sra} & \quad R_d R_s \text{imm} & \text{shift right arithmetic} & \quad \text{Reg}[R_d] \leftarrow \text{Reg}[R_s] \gg \text{imm} \\
\text{rol} & \quad R_d R_s \text{imm} & \text{rotate left} & \quad \text{Reg}[R_d] \leftarrow \text{rot(Reg}[R_s] \text{ imm left}) \\
\text{ror} & \quad R_d R_s \text{imm} & \text{rotate right} & \quad \text{Reg}[R_d] \leftarrow \text{rot(Reg}[R_s] \text{ imm right})
\end{align*}

All of these instructions can use $R_t$ instead of $\text{imm}$. 
Jump Instructions

*Jumps* control flow of program execution.

\[
\begin{align*}
\text{j } label & \quad \text{jump to location} \\
& \quad \text{PC } \leftarrow & \& \text{ label} \\
\text{jal } label & \quad \text{jump and link} \\
& \quad \text{ra } \leftarrow \text{ PC } ; \text{ PC } \leftarrow \text{ label} \\
\text{jr } R_s & \quad \text{jump via register} \\
& \quad \text{PC } \leftarrow \text{ Reg}[R_s] \\
\text{jalr } R_s & \quad \text{jump and link via reg} \\
& \quad \text{Reg}[31] \leftarrow \text{ PC } ; \text{ PC } \leftarrow \text{ Reg}[R_s]
\end{align*}
\]
Branch Instructions

Branches combine testing and jumping.

\[
\begin{align*}
\text{beq} \quad R_s & R_t \ \text{branch on equal} \\
& \quad \text{if } (\text{Reg}[R_s] == \text{Reg}[R_t]) \ PC \leftarrow \text{label} \\
\text{bne} \quad R_s & R_t \ \text{branch on not equal} \\
& \quad \text{if } (\text{Reg}[R_s] != \text{Reg}[R_t]) \ PC \leftarrow \text{label} \\
\text{blt} \quad R_s & R_t \ \text{branch on less than} \\
& \quad \text{if } (\text{Reg}[R_s] < \text{Reg}[R_t]) \ PC \leftarrow \text{label} \\
\text{ble} \quad R_s & R_t \ \text{branch on less or equal} \\
& \quad \text{if } (\text{Reg}[R_s] \leq \text{Reg}[R_t]) \ PC \leftarrow \text{label} \\
\text{bgt} \quad R_s & R_t \ \text{branch on greater than} \\
& \quad \text{if } (\text{Reg}[R_s] > \text{Reg}[R_t]) \ PC \leftarrow \text{label} \\
\text{bge} \quad R_s & R_t \ \text{branch on greater or equal} \\
& \quad \text{if } (\text{Reg}[R_s] \geq \text{Reg}[R_t]) \ PC \leftarrow \text{label}
\end{align*}
\]
## MIPS Instruction Set

Implementation of pseudo-instructions:

<table>
<thead>
<tr>
<th>What you write</th>
<th>Machine code produced</th>
</tr>
</thead>
<tbody>
<tr>
<td>li $t5, const</td>
<td>ori $t5, $0, const</td>
</tr>
<tr>
<td>la $t3, label</td>
<td>lui $at, label[31..16]</td>
</tr>
<tr>
<td></td>
<td>ori $t3, $at, label[15..0]</td>
</tr>
<tr>
<td>bge $t1, $t2, label</td>
<td>slt $at, $t1, $t2</td>
</tr>
<tr>
<td></td>
<td>beq $at, $0, label</td>
</tr>
<tr>
<td>blt $t1, $t2, label</td>
<td>slt $at, $t1, $t2</td>
</tr>
<tr>
<td></td>
<td>bne $at, $0, label</td>
</tr>
</tbody>
</table>

Note: use of $at register for intermediate results
MIPS vs SPIM

MIPS is a machine architecture, including instruction set
SPIM is an *emulator* for the MIPS instruction set

- reads text files containing instruction + directives
- converts to machine code and loads into "memory"
- provides debugging capabilities
  - single-step, breakpoints, view registers/memory, ...
- provides mechanism to interact with operating system (syscall)

Also provides extra instructions, mapped to MIPS core set

- provide convenient/mnemonic ways to do common operations
- e.g. move $s0,$v0 rather than addu $s0,$0,$v0
Using SPIM

Three ways to execute MIPS code with SPIM

• **spim** ... command line tool
  - load programs using `-file` option
  - interact using stdin/stdout via login terminal
• **qtspim** ... GUI environment
  - load programs via a load button
  - interact via a pop-up stdin/stdout terminal
• **xspim** ... GUI environment
  - similar to qtspim, but not as pretty
  - requires X-windows server
Using SPIM

Command-line tool:
Using SPIM

GUI tool:
System Calls

The SPIM interpreter provides I/O and memory allocation via the syscall instruction.

<table>
<thead>
<tr>
<th>Service</th>
<th>n</th>
<th>Arguments</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>printf(&quot;%d&quot;)</td>
<td>1</td>
<td>int in $a0</td>
<td>-</td>
</tr>
<tr>
<td>printf(&quot;%f&quot;)</td>
<td>2</td>
<td>float in $f12</td>
<td>-</td>
</tr>
<tr>
<td>printf(&quot;%lf&quot;)</td>
<td>3</td>
<td>double in $f12</td>
<td>-</td>
</tr>
<tr>
<td>printf(&quot;%s&quot;)</td>
<td>4</td>
<td>$a0 = string</td>
<td>-</td>
</tr>
<tr>
<td>scanf(&quot;%d&quot;)</td>
<td>5</td>
<td>-</td>
<td>int in $v0</td>
</tr>
<tr>
<td>scanf(&quot;%f&quot;)</td>
<td>6</td>
<td>-</td>
<td>float in $f0</td>
</tr>
<tr>
<td>scanf(&quot;%lf&quot;)</td>
<td>7</td>
<td>-</td>
<td>double in $f0</td>
</tr>
<tr>
<td>fgets</td>
<td>8</td>
<td>buffer address in $a0</td>
<td>length in $a1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>sbrk</td>
<td>9</td>
<td>nbytes in $a0</td>
<td>address in $v0</td>
</tr>
<tr>
<td>printf(&quot;%c&quot;)</td>
<td>11</td>
<td>char in $a0</td>
<td>-</td>
</tr>
<tr>
<td>scanf(&quot;%c&quot;)</td>
<td>12</td>
<td>-</td>
<td>char in $v0</td>
</tr>
<tr>
<td>exit(status)</td>
<td>17</td>
<td>status in $a0</td>
<td>-</td>
</tr>
</tbody>
</table>

Also system calls for file I/O: open, read, write, close
## MIPS (SPIM) memory layout

<table>
<thead>
<tr>
<th>Region</th>
<th>Address</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>0x004000000</td>
<td>instructions only; read-only; cannot expand</td>
</tr>
<tr>
<td>data</td>
<td>0x100000000</td>
<td>data objects; read/write; can be expanded</td>
</tr>
<tr>
<td>stack</td>
<td>0x7fffffff</td>
<td>grows down from that address; read/write</td>
</tr>
<tr>
<td>k_text</td>
<td>0x800000000</td>
<td>kernel code; read-only</td>
</tr>
<tr>
<td></td>
<td></td>
<td>only accessible in kernel mode</td>
</tr>
<tr>
<td>k_data</td>
<td>0x900000000</td>
<td>kernel data‘</td>
</tr>
<tr>
<td></td>
<td></td>
<td>only accessible in kernel mode</td>
</tr>
</tbody>
</table>
MIPS assembly language programs contain

- comments ... introduced by 
- labels ... appended with :
- directives ... symbol beginning with .
- assembly language instructions

Programmers need to specify

- data objects that live in the data region
- functions (instruction sequences) that live in the code/text region

Each instruction or directive appears on its own line
# hello.s ... print "Hello, MIPS"

main:
    la $a0, msg     # load the argument string
    li $v0, 4      # load the system call (print)
    syscall        # print the string
    jr $ra         # return to caller (__start)

.data          # the data segment
    msg: .asciiz "Hello, MIPS\n"
# Prog.s ... comment giving description of function
# Author ...

main: # indicates start of code
    # (i.e. first user instruction to execute)
    # ...

.data # variable declarations follow this line
    # ...

# End of program; leave a blank line to make SPIM happy
Assembler Directives

Directives (instructions to assembler, not MIPS instructions)

```plaintext
.text # following instructions placed in text
.data # following objects placed in data
.globl # make symbol available globally

a: .space 18 # uchar a[18]; or uint a[4];
    .align 2 # align next object on 2-byte addr

i: .word 2 # unsigned int i = 2;
v: .word 1,3,5 # unsigned int v[3] = {1,3,5};
h: .half 2,4,6 # unsigned short h[3] = {2,4,6};
b: .byte 1,2,3 # unsigned char b[3] = {1,2,3};
f: .float 3.14 # float f = 3.14;

s: .asciiz "abc" # char s[4] {'a','b','c','\0'};
t: .ascii "abc" # char s[3] {'a','b','c'};
```
Encoding MIPS Instructions as 32 bit Numbers

<table>
<thead>
<tr>
<th>Assembler</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>$7 = $8 + $0</td>
<td><code>000000sssssttttttdddddd00000100000</code></td>
</tr>
<tr>
<td>add $d, $s, $t</td>
<td><code>0000000011101000000000000000100000</code></td>
</tr>
<tr>
<td>add $7, $8, $0</td>
<td><code>0x01e80020 == 31981600</code></td>
</tr>
<tr>
<td>$5 = $1 - $3</td>
<td><code>000000sssssttttttdddddd00000100010</code></td>
</tr>
<tr>
<td>sub $d, $s, $t</td>
<td><code>00000000001000110010100000100010</code></td>
</tr>
<tr>
<td>sub $5, $1, $3</td>
<td><code>0x00232822 == 2304034</code></td>
</tr>
<tr>
<td>$2 = $2 + 1</td>
<td><code>001000ssssssddddddCCCCCCCCCCCCCCCCC</code></td>
</tr>
<tr>
<td>addi $d, $s, C</td>
<td><code>00100000010000100000000000000000000001</code></td>
</tr>
<tr>
<td>addi $2, $2, 1</td>
<td><code>0x204200000 == 541196289</code></td>
</tr>
</tbody>
</table>

All instructions are variants of 3 patterns of bits. Which simplifies silicon and human decoding (ass1!). E.g. register numbers always in same place.