Computer Fundamentals

6L for CST/NST 1A
Michaelmas 2010
MWF @ 10, Arts School “A”
Aims & Objectives

• This course aims to:
  – give you a general understanding of how a computer works
  – introduce you to assembly-level programming
  – prepare you for future courses.

• At the end of the course you’ll be able to:
  – describe the fetch-execute cycle of a computer
  – understand the different types of information which may be stored within a computer memory
  – write a simple assembly language program
Recommended Reading

• This course doesn’t follow any particular book exactly, but any of the following are useful:
    • also used in CST Part 1B “Computer Design”
  – *Digital Design and Computer Architecture*, Harris and Harris, Morgan Kaufmann 2007
    • also used in CST Part 1A “Digital Electronics”
    • good general overview book; somewhat broader in scope, and somewhat simpler to digest than above
Course Outline

• We’ll cover the following topics:
  – A Brief History of Computing
  – Operation of a Simple Computer
  – Input / Output
  – MIPS Assembly Language

• This course is new this year, but derives from Part I of pre-2010 CST 1A “Operating Systems”
  – This will help in finding e.g. past exam questions

• Feel free to ask questions during the lecture
  – or after it, or via email – see course web page
A Chronology of Early Computing

• (several BC): abacus used for counting
• 1614: logarithms discovered (John Napier)
• 1622: invention of the slide rule (Robert Bissaker)
• 1642: First mechanical digital calculator (Pascal)
• Charles Babbage (U. Cambridge) invents:
  – 1812: “Difference Engine”
  – 1833: “Analytical Engine”
• 1890: First electro-mechanical punched card data-processing machine (Hollerith)
• 1905: Vacuum tube/triode invented (De Forest)
The War Years...

- **1935**: the relay-based *IBM 601* reaches 1 MPS.
- **1939**: *ABC* - first electronic digital computer (Atanasoff & Berry)
- **1941**: *Z3* - first programmable computer (Zuse)
- Jan **1943**: the *Harvard Mark I* (Aiken)
- Dec **1943**: *Colossus* built at ‘Station X’ – Bletchley Park
- **1945**: ENIAC (Eckert & Mauchley, U. Penn):
  - 30 tons, 1000 square feet, 140 kW,
  - 18K vacuum tubes, 20×10-digit accumulators,
  - 100KHz, circa 300 MPS.
  - Used to calculate artillery firing tables.
  - (1946) blinking lights for the media... .
- But “programming” is via plug-board: tedious and slow
The Von Neumann Architecture

- **1945**: von Neumann drafts “EDVAC” report
  - design for a *stored-program* machine
  - Eckert & Mauchley mistakenly unattributed
Further Progress...

• **1947**: “point contact” transistor invented (Shockley, Bardeen & Brattain)

• **1949**: EDSAC, the world’s first stored-program computer (Wilkes & Wheeler)
  – 3K vacuum tubes, 300 square feet, 12 kW,
  – 500KHz, circa 650 IPS, 225 MPS.
  – 1024 17-bit words of memory in mercury ultrasonic delay lines – early DRAM ;-)
  – 31 word “operating system” (!)

• **1954**: TRADIC, first electronic computer without vacuum tubes (Bell Labs)
The Silicon Age

- **1954**: first silicon (junction) transistor (TI)
- **1959**: first integrated circuit (Kilby & Noyce, TI)
- **1964**: IBM System/360, based on ICs.
- **1971**: Intel 4004, first micro-processor (Ted Hoff):
  - 2300 transistors, 60 KIPS.
- **1978**: Intel 8086/8088 (used in IBM PC).
- **1980**: first VLSI chip (> 100,000 transistors)
- Today: ~800M transistors, 45nm, ~3 GHz.
Languages and Levels

- Computers programmable with variety of different languages.
  - e.g. ML, java, C/C++, python, perl, FORTRAN, Pascal, . . .
- Can describe the operation of a computer at a number of different levels; however all levels are **functionally equivalent**
- Levels relate via either (a) translation, or (b) interpretation.
Layered Virtual Machines

Consider a set of machines $M_0, M_1, \ldots, M_n$:
- Machine $M_i$ understands only machine language $L_i$
- Levels 0, -1 covered in Digital Electronics, Physics,
- Level 2 will be covered in CST 1A Operating Systems

**This course focuses on levels 1 and 3**

**NB**: all levels useful; none “the truth”.

<table>
<thead>
<tr>
<th>Level</th>
<th>Language</th>
<th>Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>High-Level Language, e.g. ML</td>
<td></td>
<td>Virtual Machine M5 (Language L5)</td>
</tr>
<tr>
<td>Compiled Language (e.g. C++)</td>
<td></td>
<td>Virtual Machine M4 (Language L5)</td>
</tr>
<tr>
<td>Assembly Language Programs</td>
<td></td>
<td>Virtual Machine M3 (Language L3)</td>
</tr>
<tr>
<td>Operating System Level</td>
<td></td>
<td>Virtual Machine M2 (Language L2)</td>
</tr>
<tr>
<td>Computer Organization Level</td>
<td></td>
<td>Virtual Machine M1 (Language L1)</td>
</tr>
<tr>
<td>Digital Logic Level</td>
<td></td>
<td>“Actual” Machine M0 (Language L0)</td>
</tr>
</tbody>
</table>
Digital Electronics in a Slide

• Take an electric circuit but treat “high” voltages as 1, and “low” voltages as 0
• Using *transistors*, can build *logic gates*
  – Deterministic functions of inputs (1s and 0s)
• Circuit diagrams use symbols as short hand, e.g.

![Logic Gate Diagrams](image)

- Output is ‘1’ only if **both** inputs are ‘1’
- Output is ‘1’ if **either** input is ‘1’
- Output is ‘1’ only if input is ‘0’
- Output is ‘1’ only if inputs are **different**

• Using *feedback* (outputs become inputs) we can build other stuff (latches, flip-flops, ...)
• Low-level circuit diagrams are not examinable
A (Simple) Modern Computer

![Diagram of a modern computer system](image)

**Processor**
- Register File (including PC)
  - Control Unit
  - Execution Unit

**Bus**
- Address Data Control

**Memory**
- e.g. 1 GByte
  - $2^{30} \times 8 = 8,589,934,592$ bits

**Connections**
- Hard Disk
- Framebuffer
- Sound Card
- Super I/O
  - Mouse
  - Keyboard
  - Serial
A (Simple) Modern Computer

Processor (CPU): executes programs

Memory: stores programs & data

Bus: connects everything together

Devices: for input and output

Register File (including PC)

Control Unit

Execution Unit

Hard Disk

Super I/O

Mouse

Keyboard

Serial

Framebuffer

Sound Card

e.g. 1 GByte

$2^{30} \times 8 = 8,589,934,592$ bits
Computers all about operating on information:
  – information arrives into memory from input devices
  – memory is a large “byte array” which can hold anything we want

Computer conceptually takes values from memory, performs whatever operations, and then stores results back

In practice, CPU operates on registers:
  – a register is an extremely fast piece of on-chip memory
  – modern CPUs have between 8 and 128 registers, each 32/64 bits
  – data values are loaded from memory into registers before operation
  – result goes into register; eventually stored back to memory again.
Memory Hierarchy

- Use **cache** between main memory & registers to hide “slow” DRAM.
- Cache made from faster SRAM: more expensive, and hence smaller.  
  - holds copy of subset of main memory.
- Split of instruction and data at cache level:  
  - “Harvard” architecture.
- Cache <-> CPU interface uses a custom bus.
- Today have ~8MB cache, ~4GB RAM.
Static RAM (SRAM)

- Relatively fast (currently 5 – 20\text{ns}).
- This is the \textit{Digital Logic} view:
  - Some wires, some gates, and some “D-latches”
Static RAM (SRAM)

- Relatively fast (currently 5 – 20ns).
- This is the Digital Logic view:
  - Some wires, some gates, and some “D-latches”
SRAM Reality

• Data held in cross-coupled inverters.

• One **word** line, two **bit** lines.

• **To read:**
  – precharge both bit and $\bar{\text{bit}}$, and then strobe word
  – $\bar{\text{bit}}$ discharged if there was a 1 in the cell;
  – bit discharged if there was a 0.

• **To write:**
  – precharge either bit (for “1”) or $\bar{\text{bit}}$ (for “0”),
  – strobe word.
Dynamic RAM (DRAM)

- Use a **single transistor** to store a bit.
- **Write**: put value on bit lines, strobe word line.
- **Read**: pre-charge, strobe word line, amplify, latch.
- “Dynamic”: refresh periodically to restore charge.
- Slower than SRAM: typically 50ns – 100ns.
• Two stage: row, then column.
• Usually share address pins: RAS & CAS select decoder or mux.
• FPM, EDO, SDRAM faster for same row reads.
The Fetch-Execute Cycle

- A special register called **PC** holds a memory address
  - on reset, initialized to 0.

- Then:
  1. Instruction *fetched* from memory address held in **PC** into instruction buffer (**IB**)
  2. Control Unit determines what to do: *decodes* instruction
  3. Execution Unit *executes* instruction
  4. **PC** updated, and back to Step 1

- Continues pretty much forever...
The Execution Unit

- The “calculator” part of the processor.
- Broken into parts (functional units), e.g.
  - Arithmetic Logic Unit (ALU).
  - Shifter/Rotator.
  - Multiplier.
  - Divider.
  - Memory Access Unit (MAU).
  - Branch Unit.
- Choice of functional unit determined by signals from control unit.
Arithmetic Logic Unit (ALU)

• Part of the execution unit.
• Inputs from register file; output to register file.
• Performs simple two-operand functions:
  – $a + b$; $a - b$; $a$ AND $b$; $a$ OR $b$; etc
• Typically perform all possible functions; use function code to select (mux) output.
Number Representation

- n-bit register $b_{n-1}b_{n-2} \ldots b_1b_0$ can represent $2^n$ different values.
- Call $b_{n-1}$ the **most significant bit** (msb), $b_0$ the **least significant bit** (lsb).
- Unsigned numbers: $val = b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \cdots + b_12^1 + b_02^0$
  - e.g. $1101_2 = 2^3 + 2^2 + 2^0 = 8 + 4 + 1 = 13$.
- Represents values from 0 to $2^{n-1}$ inclusive.
- For large numbers, binary is unwieldy: use hexadecimal (base 16).
- To convert, group bits into groups of 4, e.g.
  - $111101010_2 = 0011|1110|1010_2 = 3EA_{16}$.
- Often use “0x” prefix to denote hex, e.g. 0x107.
- Can use dot to separate large numbers into 16-bit chunks, e.g.
  - 0x3FF.FFFF
Signed Numbers

• What about signed numbers? Two main options:
  • **Sign & magnitude:**
    – top (leftmost) bit flags if negative; remaining bits make value.
    – e.g. byte $10011011_2 \rightarrow -0011011_2 = -27$.
    – represents range $-(2^{n-1} - 1)$ to $+(2^{n-1} - 1)$ ...
    – ... and the bonus value $-0 (!)$
  • **2’s complement:**
    – to get $-x$ from $x$, invert every bit and add 1.
    – e.g. $+27 = 00011011_2 \Rightarrow -27 = (11100100_2 + 1) = 11100101_2$.
    – treat $1000 \ldots 000_2$ as $-2^{n-1}$
    – represents range $-2^{n-1}$ to $+(2^{n-1} - 1)$
  • **Note:**
    – in both cases, top-bit means “negative”.
    – both representations depend on $n$;
  • In practice, all modern computers use 2’s complement...
Unsigned Arithmetic

• Unsigned addition (using 5-bit registers)

\[
\begin{array}{cccc}
0 & 1 & 1 & 1 \\
+ 0 & 0 & 1 & 1 \\
\hline
0 & 0 & 1 & 1
\end{array}
\]

\[
\begin{array}{cccc}
1 & 1 & 1 & 0 \\
+ 0 & 0 & 1 & 1 \\
\hline
1 & 0 & 0 & 1
\end{array}
\]

• Carry bits \( C_0 (=C_{\text{in}}), \ C_1, \ C_2, \ldots \ C_n (=C_{\text{out}}) \)
  – usually refer to \( C_n \) as \( C \), the \textit{carry flag}
  – In addition, if \( C \) is 1, we got the wrong answer

• Unsigned subtraction: if \( C \) is 0, we “borrowed”

\[
\begin{array}{cccc}
1 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 \\
+ 0 & 0 & 1 & 0 \\
\hline
1 & 0 & 0 & 0
\end{array}
\]

\[
\begin{array}{cccc}
0 & 1 & 1 & 0 \\
0 & 0 & 1 & 1 \\
+ 1 & 0 & 1 & 1 \\
\hline
0 & 1 & 1 & 1
\end{array}
\]

Wrong! (by \( 32=2^5 \))
Wrong! (again by \( 2^5 \))
Signed Arithmetic

- In signed arithmetic, \( C \) on its own is useless...
  - Instead use **overflow flag**, \( V = C_n \oplus C_{n-1} \)

\[
\begin{array}{c}
01110 \\
00101 \\
+ 00111 \\
\hline
001100
\end{array} \quad 5
\begin{array}{c}
11100 \\
01010 \\
+ 00111 \\
\hline
010001
\end{array} \quad 12
\]

\( C_n \) and \( C_{n-1} \) are different \( \Rightarrow V=1 \)

\[
\begin{array}{c}
10000 \\
01010 \\
+ 11001 \\
\hline
100011
\end{array} \quad 3
\begin{array}{c}
01100 \\
10110 \\
+ 10110 \\
\hline
101100
\end{array} \quad -10
\]

\( C \) is set...

\( \ldots \) but answer is correct

- **Negative flag** \( N = C_{n-1} \) (i.e. msb) flips on overflow

\[
\begin{array}{c}
11100 \\
01010 \\
+ 00111 \\
\hline
010011
\end{array} \quad 15
\begin{array}{c}
01100 \\
10110 \\
+ 10110 \\
\hline
101100
\end{array} \quad 10
\]

\( V=1 \Rightarrow \text{wrong} \)

Wrong by 32=2^5
Arithmetic and Logical Instructions

- Both $d$ and $a$ must be registers; $b$ can be a register or, in most machines, can also be a (small) constant.
- Typically also have **addc** and **subc**, which handle carry or borrow (for multi-precision arithmetic), e.g.
  
  ```
  add d0, a0, b0 // compute "low" part
  addc d1, a1, b1 // compute "high" part
  ```

- May also get:
  - Arithmetic shifts: **asr** and **asl** (?)
  - Rotates: **ror** and **rol**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>C/Java Equivalent</th>
<th>Mnemonic</th>
<th>C/Java Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>and $d \leftarrow a, b$</td>
<td>$d = a &amp; b$</td>
<td>add $d \leftarrow a, b$</td>
<td>$d = a + b$</td>
</tr>
<tr>
<td>xor $d \leftarrow a, b$</td>
<td>$d = a \lor b$</td>
<td>sub $d \leftarrow a, b$</td>
<td>$d = a - b$</td>
</tr>
<tr>
<td>orr $d \leftarrow a, b$</td>
<td>$d = a \lor b$</td>
<td>rsb $d \leftarrow a, b$</td>
<td>$d = b - a$</td>
</tr>
<tr>
<td>bis $d \leftarrow a, b$</td>
<td>$d = a \lor b$</td>
<td>shl $d \leftarrow a, b$</td>
<td>$d = a &lt;&lt; b$</td>
</tr>
<tr>
<td>bic $d \leftarrow a, b$</td>
<td>$d = a &amp; \neg b$</td>
<td>shr $d \leftarrow a, b$</td>
<td>$d = a &gt;&gt; b$</td>
</tr>
</tbody>
</table>
1-bit ALU Implementation

- 8 possible functions:
  1. \( a \text{ AND } b, \ a \text{ AND } \overline{b} \)
  2. \( a \text{ OR } b, \ a \text{ OR } \overline{b} \)
  3. \( a + b, \ a + b \text{ with carry} \)
  4. \( a - b, \ a - b \text{ with borrow} \)
- To make n-bit ALU bit, connect together (use carry-lookahead on adders)
Conditional Execution

• Seen C,N,V flags; now add Z (zero), logical NOR of all bits in output.
• Can predicate execution based on (some combination) of flags, e.g.

```c
subs d, a, b    // compute d = a - b
beq proc1      // if equal, goto proc1
br proc2        // otherwise goto proc2
```

– Java equivalent approximately:
  ```java
  if (a==b) proc1() else proc2();
  ```

• On most computers, mainly limited to branches; but on ARM (and IA64), everything conditional, e.g.

```c
sub   d, a, b    // compute d = a - b
moveq d, #5     // if equal, d = 5;
movne d, #7      // otherwise d = 7;
```

– Java equivalent:
  ```java
d = (a==b) ? 5 : 7;
  ```

• “Silent” versions useful when don’t really want result, e.g. teq, cmp
## Condition Codes

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Meaning</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ, Z</td>
<td>Equal, zero</td>
<td>Z == 1</td>
</tr>
<tr>
<td>NE, NZ</td>
<td>Not equal, non-zero</td>
<td>Z == 0</td>
</tr>
<tr>
<td>MI</td>
<td>Negative</td>
<td>N == 1</td>
</tr>
<tr>
<td>PL</td>
<td>Positive (incl. zero)</td>
<td>N == 0</td>
</tr>
<tr>
<td>CS, HS</td>
<td>Carry, higher or same</td>
<td>C == 1</td>
</tr>
<tr>
<td>CC, LO</td>
<td>No carry, lower</td>
<td>C == 0</td>
</tr>
<tr>
<td>HI</td>
<td>Higher</td>
<td>C == 1 &amp;&amp; Z == 0</td>
</tr>
<tr>
<td>LS</td>
<td>Lower or same</td>
<td>C == 0</td>
</tr>
<tr>
<td>VS</td>
<td>Overflow</td>
<td>V == 1</td>
</tr>
<tr>
<td>VC</td>
<td>No overflow</td>
<td>V == 0</td>
</tr>
<tr>
<td>GE</td>
<td>Greater than or equal</td>
<td>N == V</td>
</tr>
<tr>
<td>GT</td>
<td>Greater than</td>
<td>N == V &amp;&amp; Z == 0</td>
</tr>
<tr>
<td>LT</td>
<td>Less than</td>
<td>N != V</td>
</tr>
<tr>
<td>LE</td>
<td>Less than or equal</td>
<td>N != V</td>
</tr>
</tbody>
</table>

Used to compare **unsigned** numbers (recall C==0 means we borrowed)

Used to compare **signed** numbers (note must check both N and V)
Loads and Stores

- Have variable sized values, e.g. bytes (8-bits), words (16-bits), longwords (32-bits) and quadwords (64-bits).
- Load or store instructions usually have a suffix to determine the size, e.g. ‘b’ for byte, ‘w’ for word, ‘l’ for longword.
- When storing > 1 byte, have two main options: big endian and little endian; e.g. storing 0xDEADBEEF into memory at address 0x4

<table>
<thead>
<tr>
<th>Big Endian</th>
<th>Little Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 01 02 03</td>
<td>01 02 03 04</td>
</tr>
<tr>
<td>EF BE AD DE</td>
<td>EF BE AD DE</td>
</tr>
</tbody>
</table>

- If read back a byte from address 0x4, get 0xDE if big-endian, or 0xEF if little-endian.
  - If you always load and store things of the same size, things are fine.
- Today have x86 little-endian; Sparc big-endian; Mips & ARM either.
- Annoying. . . and burns a considerable number of CPU cycles on a daily basis. . .
Accessing Memory

• To load/store values need the **address** in memory.
• Most modern machines are byte addressed: consider memory a big array of $2^A$ bytes, where $A$ is the number of address lines in the bus.
• Lots of things considered “memory” via address decoder, e.g.

![Address Decoder Diagram]

<table>
<thead>
<tr>
<th>Device</th>
<th>Size</th>
<th>Data</th>
<th>Decodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>UART</td>
<td>256 bytes</td>
<td>8-bit</td>
<td>A[0:7]</td>
</tr>
</tbody>
</table>

• Typically devices decode only a subset of low address lines, e.g.
Addressing Modes

• An *addressing mode* tells the computer where the data for an instruction is to come from.

• Get a wide variety, e.g.
  - Register: `add r1, r2, r3`
  - Immediate: `add r1, r2, #25`
  - PC Relative: `beq 0x20`
  - Register Indirect: `ldr r1, [r2]`
  - " + Displacement: `str r1, [r2, #8]`
  - Indexed: `movl r1, (r2, r3)`
  - Absolute/Direct: `movl r1, $0xF1EA0130`
  - Memory Indirect: `addl r1, ($0xF1EA0130)`

• Most modern machines are load/store ⇒ only support first five:
  - allow at most one memory ref per instruction
  - (there are very good reasons for this)

• Note that CPU generally doesn’t care *what* is being held within the memory – up to *programmer* to interpret whether data is an integer, a pixel or a few characters in a novel...
Representing Text

- Two main standards:
  1. **ASCII**: 7-bit code holding (English) letters, numbers, punctuation and a few other characters.
  2. **Unicode**: 16-bit code supporting practically all international alphabets and symbols.

- ASCII default on many operating systems, and on the early Internet (e.g. e-mail).
- Unicode becoming more popular (especially UTF-8!)
- In both cases, represent in memory as either *strings* or *arrays*: e.g. “Pub Time!” in ASCII:

```
String

<table>
<thead>
<tr>
<th></th>
<th>20</th>
<th>62</th>
<th>75</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>65</td>
<td>6D</td>
<td>69</td>
<td>54</td>
</tr>
<tr>
<td>xx</td>
<td></td>
<td>xx</td>
<td>00</td>
<td>21</td>
</tr>
</tbody>
</table>
```

```
Array

<table>
<thead>
<tr>
<th></th>
<th>75</th>
<th>50</th>
<th>00</th>
<th>09</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>69</td>
<td>54</td>
<td>20</td>
<td>62</td>
</tr>
<tr>
<td>xx</td>
<td></td>
<td>21</td>
<td>65</td>
<td>6D</td>
</tr>
</tbody>
</table>
```

N (here 2) bytes hold length, followed by characters

Byte per character, terminated with 0
Floating Point

• In many cases need very large or very small numbers
• Use idea of “scientific notation”, e.g. $n = m \times 10^e$
  – $m$ is called the mantissa
  – $e$ is called the exponent.
  e.g. $C = 3.01 \times 10^8$ m/s.

• For computers, use binary i.e. $n = m \times 2^e$, where $m$ includes a “binary point”.
• Both $m$ and $e$ can be positive or negative; typically
  – sign of mantissa given by an additional sign bit, $s$
  – exponent is stored in a biased (excess) format

$\Rightarrow$ use $n = (-1)^s m \times 2^{e-b}$, where $0 \leq m < 2$, and $b$ is the bias
• e.g. with a 4-bit mantissa and a 3-bit bias-3 exponent, you can represent positive range $[0.001_2 \times 2^{-3}, 1.111_2 \times 2^4]$
  $= [ (1/8)(1/8), (15/8)(16) ] = [ 1/64, 30 ]$
IEEE Floating Point

- To avoid redundancy, in practice modern computers use IEEE floating point with \textit{normalised} mantissa \( m = 1.xx \ldots x_2 \)
  \[ n = (-1)^s((1 + m) \times 2^{e-b}) \]
- Both single precision (32 bits) and double precision (64 bits)

| IEEE fp reserves \( e = 0 \) and \( e = \text{max} \):
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>±0 (!): both ( e ) and ( m ) zero.</td>
</tr>
<tr>
<td>±( \infty ): ( e = \text{max}, m ) zero.</td>
</tr>
<tr>
<td>NaNs: ( e = \text{max}, m ) non-zero.</td>
</tr>
<tr>
<td>denorms: ( e = 0, m ) non-zero</td>
</tr>
</tbody>
</table>
- Normal positive range \([2^{-126}, \sim 2^{128}]\) for single, or \([2^{-1022}, \sim 2^{1024}]\) for double precision.
- \textbf{NB}: still only \(2^{32}/2^{64}\) values — just spread out.
Data Structures

• Records / structures: each field stored as an offset from a **base address**

• Variable size structures: explicitly store addresses (**pointers**) inside structure, e.g.

```
datatype rec = node of int * int * rec
  | leaf of int;
val example = node(4, 5, node(6, 7, leaf(8)));
```

• Imagine **example** is stored at address 0x1000:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0F30</td>
<td>0xFFFF</td>
<td>Constructor tag for a leaf</td>
</tr>
<tr>
<td>0x0F34</td>
<td>8</td>
<td>Integer 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;leaf&quot; tag says we’re done...</td>
</tr>
<tr>
<td>0x0F3C</td>
<td>0xFFFE</td>
<td>Constructor tag for a node</td>
</tr>
<tr>
<td>0x0F40</td>
<td>6</td>
<td>Integer 6</td>
</tr>
<tr>
<td>0x0F44</td>
<td>7</td>
<td>Integer 7</td>
</tr>
<tr>
<td>0x0F48</td>
<td>0x0F30</td>
<td>Address of inner node</td>
</tr>
<tr>
<td>0x1000</td>
<td>0xFFFE</td>
<td>Constructor tag for a node</td>
</tr>
<tr>
<td>0x1004</td>
<td>4</td>
<td>Integer 4</td>
</tr>
<tr>
<td>0x1008</td>
<td>5</td>
<td>Integer 5</td>
</tr>
<tr>
<td>0x100C</td>
<td>0x0F3C</td>
<td>Address of inner node</td>
</tr>
</tbody>
</table>
### Instruction Encoding

- An instruction comprises:
  a. an **opcode**: specifies what to do.
  b. zero or more **operands**: where to get values

- Old machines (and x86) use variable length encoding for low code density; most other modern machines use fixed length encoding for simplicity, e.g. **ARM ALU instructions**:

```
<table>
<thead>
<tr>
<th>Cond</th>
<th>00</th>
<th>I</th>
<th>Opcode</th>
<th>S</th>
<th>Ra</th>
<th>Rd</th>
<th>Operand 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0011</td>
<td>00</td>
<td>0</td>
<td>1010</td>
<td>1</td>
<td>0001</td>
<td>000</td>
<td>000000000010</td>
</tr>
<tr>
<td>1110</td>
<td>00</td>
<td>0</td>
<td>1110</td>
<td>0</td>
<td>0011</td>
<td>0011</td>
<td>000000000010</td>
</tr>
<tr>
<td>1110</td>
<td>00</td>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>1101</td>
<td>1101</td>
<td>000011111111</td>
</tr>
</tbody>
</table>
```

- **and r13, r13, #255**
- **bic r03, r03, r02**
- **cmp r01, r02**
Fetch-Execute Cycle Revisited

1. CU fetches & decodes instruction and generates (a) control signals and (b) operand information.
2. In EU, control signals select functional unit ("instruction class") and operation.
3. If ALU, then read 1–2 registers, perform op, and (probably) write back result.
4. If BU, test condition and (maybe) add value to PC.
5. If MAU, generate address ("addressing mode") and use bus to read/write value.
6. Repeat ad infinitum
A (Simple) Modern Computer

- Processor
  - Register File (including PC)
  - Control Unit
  - Execution Unit

- Bus
  - Address Data Control

- Memory
  - e.g. 1 GByte
  - $2^{30} \times 8 = 8,589,934,592$ bits

- Devices: for input and output
  - Hard Disk
  - Framebuffer
  - Sound Card

- Reset
  - Mouse
  - Keyboard
  - Serial
Input/Output Devices

• Devices connected to processor via a bus (e.g. PCI)
• Includes a wide range:
  – Mouse,
  – Keyboard,
  – Graphics Card,
  – Sound card,
  – Floppy drive,
  – Hard-Disk,
  – CD-Rom,
  – Network card,
  – Printer,
  – Modem
  – etc.
• Often two or more stages involved (e.g. USB, IDE, SCSI, RS-232, Centronics, etc.)
UARTs

- **UART** = Universal Asynchronous Receiver/Transmitter:
  - stores 1 or more bytes internally
  - converts parallel to serial
  - outputs according to RS-232
- Various baud rates (e.g. 1,200 – 115,200)
- Slow and simple. . . and very useful.
- Make up “serial ports” on PC
- Max throughput 14.4KBytes; variants up to 56K (for modems).
Hard Disks

- Whirling bits of (magnetized) metal...
- Bit like a double-sided record player: but rotates 3,600–12,000 times a minute ;-)
- **To read/write data:**
  - *move arms to cylinder*
  - *wait for sector*
  - *activate head*
- Today capacities are around \( \sim 500 \text{ GBytes} \) (=500 \( \times \) 2\(^{30}\) bytes)
Graphics Cards

• Essentially some RAM (**framebuffer**) and some digital-to-analogue circuitry (**RAMDAC**) – latter only required for CRTs
• (Today usually also have powerful GPU for 3D)
• Framebuffer holds 2-D array of **pixels**: picture elements.
• Various **resolutions** (640x480, 1280x1024, etc) and **color depths**: 8-bit (LUT), 16-bit (RGB=555), 24-bit (RGB=888), 32-bit (RGBA=888)
• Memory requirement = \(x \times y \times \text{depth}\)
• e.g. 1280x1024 @ 32bpp needs 5,120KB for screen
• => full-screen 50Hz video requires 250 MBytes/s (or 2Gbit/s!)
Buses

- Bus = a collection of *shared* communication wires:
  - ✓ low cost
  - ✓ versatile / extensible
  - ✗ potential bottle-neck
- Typically comprises *address lines, data lines* and *control lines*
  - and of course power/ground
- Operates in a *master-slave* manner, e.g.
  1. master decides to e.g. read some data
  2. master puts address onto bus and asserts ‘read’
  3. slave reads address from bus and retrieves data
  4. slave puts data onto bus
  5. master reads data from bus
In practice, have lots of different buses with different characteristics e.g. data width, max #devices, max length.

Most buses are **synchronous** (share clock signal).
Synchronous Buses

Figure shows a read transaction which requires three bus cycles

1. CPU puts addr onto address lines and, after settle, asserts control lines.
2. Device (e.g. memory) fetches data from address.
3. Device puts data on data lines, CPU latches value and then finally deasserts control lines.
   - If device not fast enough, can insert *wait states*
   - Faster clock/longer bus can give *bus skew*
Asynchronous Buses

- Asynchronous buses have no shared clock; instead use **handshaking**, e.g.
  - CPU puts address onto address lines and, after settle, asserts control lines
  - next, CPU asserts `/SYN` to say everything ready
  - once memory notices `/SYN`, it fetches data from address and puts it onto bus
  - memory then asserts `/ACK` to say data is ready
  - CPU latches data, then deasserts `/SYN`
  - finally, Memory deasserts `/ACK`

- More handshaking if multiplex address & data lines
Interrupts

- Bus reads and writes are *transaction* based: CPU requests something and waits until it happens.
- But e.g. reading a block of data from a hard-disk takes ~2ms, which might be over 10,000,000 clock cycles!
- **Interrupts** provide a way to decouple CPU requests from device responses.
  1. CPU uses bus to make a request (e.g. writes some special values to addresses decoded by some device).
  2. Device goes off to get info.
  3. Meanwhile CPU continues doing other stuff.
  4. When device finally has information, raises an *interrupt*.
  5. CPU uses bus to read info from device.
- When interrupt occurs, CPU *vectors* to handler, then *resumes* using special instruction, e.g.

```
0x184c: add r0, r0, #8
0x1850: sub r1, r5, r6
0x1854: ldr r0, [r0]
0x1858: and r1, r1, r0
```

```
0x0020: ...
0x0024: <do stuff>
..... ...
0x0038: rti
```
Interrupts (2)

• Interrupt lines (~4–8) are part of the bus.
• Often only 1 or 2 pins on chip ⇒ need to encode.
• e.g. ISA & x86:

1. Device asserts $\text{IR}_x$
2. PIC asserts $\text{INT}$
3. When CPU can interrupt, strobes $\text{INTA}$
4. PIC sends interrupt number on $\text{D}[0:7]$
5. CPU uses number to index into a table in memory which holds the addresses of handlers for each interrupt.
6. CPU saves registers and jumps to handler
Direct Memory Access (DMA)

• Interrupts are good, but even better is a device which can read and write processor memory \textit{directly}.
• A generic DMA “command” might include
  – source address
  – source increment / decrement / do nothing
  – sink address
  – sink increment / decrement / do nothing
  – transfer size
• Get one interrupt at end of data transfer
• DMA channels may be provided by devices themselves:
  – e.g. a disk controller
  – pass disk address, memory address and size
  – give instruction to read or write
• Also get “stand-alone” programmable DMA controllers.
Computer Organization: Summary

• Computers made up of four main parts:
  1. Processor (including register file, control unit and execution unit – with ALU, memory access unit, branch unit, etc),
  2. Memory (caches, RAM, ROM),
  3. Devices (disks, graphics cards, etc.), and
  4. Buses (interrupts, DMA).

• Information represented in all sorts of formats:
  – signed & unsigned integers,
  – strings,
  – floating point,
  – data structures,
  – instructions.

• Can (hopefully) understand all of these at some level, but gets pretty complex...

• Next up: bare bones programming with MIPS assembly...
What is MIPS?

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

• Also used by big-iron SGI machines (R1x000)
MIPS Instructions

- MIPS has 3 instruction formats:
  - **R-type** - register operands
  - **I-type** - immediate operands
  - **J-type** - jump operands

- All instructions are 1 word long (32 bits)

- Examples of R-type instructions:
  
  ```
  add $8, $1, $2  # $8 <= $1 + $2
  sub $12, $6, $3  # $12 <= $6 - $3
  and $1, $2, $3  # $1 <= $2 & $3
  or $1, $2, $3   # $1 <= $2 | $3
  ```

- Register 0 ($0) always contains zero
  
  ```
  add $8, $0, $0  # $8 <= 0
  add $8, $1, $0  # $8 <= $1
  ```
R-Type Instructions

- These take three register operands ($0 .. $31)
- R-type instructions have six fixed-width fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rs</td>
<td>the first register source operand</td>
</tr>
<tr>
<td>Rt</td>
<td>the second register source operand</td>
</tr>
<tr>
<td>Rd</td>
<td>the register destination operand; gets result of the operation</td>
</tr>
<tr>
<td>shamt</td>
<td>shift amount (0 if not shift instruction)</td>
</tr>
<tr>
<td>funct</td>
<td>This field selects the specific variant of the operation and is sometimes called the <em>function code</em>; e.g. for opcode 0, if (funct == 32) =&gt; <strong>add</strong> ; if (funct == 34) =&gt; <strong>sub</strong></td>
</tr>
</tbody>
</table>

- **opcode** basic operation of the instruction

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rs</td>
<td>5</td>
</tr>
<tr>
<td>Rt</td>
<td>5</td>
</tr>
<tr>
<td>Rd</td>
<td>5</td>
</tr>
<tr>
<td>shamt</td>
<td>5</td>
</tr>
<tr>
<td>funct</td>
<td>6</td>
</tr>
</tbody>
</table>

- **opcode** 6 bits
- **Rs** 5 bits
- **Rt** 5 bits
- **Rd** 5 bits
- **shamt** 5 bits
- **funct** 6 bits
I-Type Instructions

- I = **Immediate**
  - Value is encoded in instruction & available directly
  - MIPS allows 16-bit values (only 12-bits on ARM)
- Useful for loading constants, e.g:
  - `li $7, 12 # load constant 12 into reg7`
- This is a big win in practice since >50% of arithmetic instructions involve constants!
- MIPS supports several immediate mode instructions: **opcode** determines which one.
Immediate Addressing on MIPS

- **or**, **and**, **xor** and **add** instructions have immediate forms which take an “**i**” suffix, e.g:

  ```
  ori $8, $0, 0x123  # puts 0x00000123 into r8
  ori $9, $0, -6    # puts 0x0000fffaf into r9
  addi $10, $0, 0x123 # puts 0x00000123 into r10
  addi $11, $0, -6  # puts 0xfffffffa into r11
  # (note sign extension...)
  ```

- **lui** instruction loads upper 16 bits with a constant and sets the least-significant 16 bits to zero

  ```
  lui $8, 0xabcd     # puts 0xabcd0000 into r8
  ori $8, $0, 0x123  # sets just low 16 bits
  # result: r8 = 0xabcd0123
  ```

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

• Last instruction format: **Jump-type** (J-Type)

![Instruction Format Diagram]

- **opcode**
- **target address (in #instructions)**

- Only used by unconditional jumps, e.g.
  
  ```
  j dest_addr  \# jump to (target<<2)
  ```

- Cannot directly jump more than $2^{26}$ instructions away (see later...)

• **Branches** use I-type, not J-type, since must specify 2 registers to compare, e.g.

  ```
  beq $1, $2, dest \# goto dest iff $1==$2
  ```
Big Picture

\[ x = a - b + c - d; \]

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

\[
\begin{array}{cccccc}
0 & 4 & 5 & 10 & 0 & 34 \\
0 & 6 & 7 & 11 & 0 & 34 \\
0 & 10 & 11 & 12 & 0 & 32 \\
\end{array}
\]

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

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

- Registers are used for specific purposes, by *convention*
- For example, registers 4, 5, 6 and 7 are used as *parameters* or *arguments* for subroutines (see later)
- Can be specified as $4, $5, $6, $7 or as $a0, $a1, $a2 and $a3
- Other examples:

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>$0</td>
<td>zero</td>
</tr>
<tr>
<td>$at</td>
<td>$1</td>
<td>assembler temporary</td>
</tr>
<tr>
<td>$v0, $v1</td>
<td>$2, $3</td>
<td>expression eval &amp; result</td>
</tr>
<tr>
<td>$t0...$t7</td>
<td>$8...$15</td>
<td>temporary registers</td>
</tr>
<tr>
<td>$s0...$s7</td>
<td>$16...$23</td>
<td>saved temporaries</td>
</tr>
<tr>
<td>$t8, $t9</td>
<td>$24, $25</td>
<td>temporary</td>
</tr>
<tr>
<td>$k0, $k1</td>
<td>$26, $27</td>
<td>kernel temporaries</td>
</tr>
<tr>
<td>$gp</td>
<td>$28</td>
<td>global pointer</td>
</tr>
<tr>
<td>$sp</td>
<td>$29</td>
<td>stack pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>$30</td>
<td>frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>$31</td>
<td>return address</td>
</tr>
</tbody>
</table>
Our first program: Hello World!

```
.text       # begin code section
.globl main
main:      li $v0, 4    # system call for print string
           la $a0, str  # load address of string to print
           syscall     # print the string
           li $v0, 10  # system call for exit
           syscall     # exit

.data       # begin data section
str:        .asciiz "Hello world!\n"
            # NUL terminated string, as in C
```

• Comments (after "#") to aid readability
• Assembly language 5-20x line count of high level languages
• (And empirical wisdom is that development time strongly related to number of lines of code...)
Assembler Directives

- On previous slide saw various things that weren’t assembly code instructions: *labels* and *directives*
- These are here to assist assembler to do its job ...
- ... but do not necessarily produce results in memory
- Examples:
  ```
  main:              tell assembler where program starts
  str:               user-friendly[er] way to refer to a memory address
  
  .text              tells assembler that following is part of code area
  .data               following is part of data area
  .ascii str          insert ASCII string into next few bytes of memory
  .asciiz str         ...as above, but add null byte at end
  .word n1,n2         reserve space for words and store values n1, n2 etc. in them
  .half n1,n2         reserve space for halfwords and store values n1, n2 in them
  .byte n1,n2         reserve space for bytes and store values n1, n2 in them
  .space n            reserve space for n bytes
  .align m            align the next datum on $2^m$ byte boundary, e.g. .align 2 aligns on word boundary
  ```
Pseudo Instructions

- Assemblers can also support other things that look like assembly instructions... but aren’t!
  - These are called **pseudo-instructions** and are there to make life easier for the programmer
  - Can be built from other actual instructions

- Some examples are:

<table>
<thead>
<tr>
<th>Pseudo Instruction</th>
<th>Translated to</th>
</tr>
</thead>
<tbody>
<tr>
<td>move $1,$2</td>
<td>add $1, $0, $2</td>
</tr>
<tr>
<td>li $1, 678</td>
<td>ori $1, $0, 678</td>
</tr>
<tr>
<td>la $8, 6($1)</td>
<td>addi $8, $1, 6</td>
</tr>
<tr>
<td>la $8, label</td>
<td>lui $1, label[31:16]</td>
</tr>
<tr>
<td>b label</td>
<td>bgez $0, $0, label</td>
</tr>
<tr>
<td>beq $8, 66, label</td>
<td>ori $1, $0, 66</td>
</tr>
<tr>
<td></td>
<td>beq $1, $8, label</td>
</tr>
</tbody>
</table>
Accessing Memory (Loads & Stores)

• Can load **bytes, half-words, or words**
  \[ \text{lb} \] \ $a0, c($s1) \quad \# \text{load byte}; \ a0 = \text{Mem}[s1+c] \\
  \[ \text{lh} \] \ $a0, c($s1) \quad \# \text{load half-word [16 bits]} \\
  \[ \text{lw} \] \ $a0, c($s1) \quad \# \text{load word [32 bits]} \\
  \text{– gets data from memory and puts into a register} \\
  \text{– c is a [small] constant; can omit if zero} \\
• Same for stores using **sb, sh, and sw**
• lw, sw etc are **l-type** instructions:
  \text{– destination register ($a0), source register ($s1), and} \\
  \text{16-bit immediate value (constant c)} \\
• However assembler also allows lw/sw (and la) \\
  \text{to be pseudo-instructions e.g.} \\
  \[ \text{lw} \] \ $a0, \text{addr} \quad \rightarrow \quad \text{lui} \ $1, \text{addr}[31:16] \\
  \text{lw} \ $a0, \text{addr}[15:0](\text{$1$})
Control Flow Instructions

Assembly language has very few control structures...

- **Branch instructions**: if \(<\text{cond}\) then goto \(<\text{label}\>

\[
\begin{align*}
\text{beqz} & \quad \text{if } s0 == 0 \quad \text{goto label} \\
\text{bnez} & \quad \text{if } s0 != 0 \quad \text{goto label} \\
\text{bge} & \quad \text{if } s0 \geq s1 \quad \text{goto label} \\
\text{ble} & \quad \text{if } s0 \leq s1 \quad \text{goto label} \\
\text{blt} & \quad \text{if } s0 < s1 \quad \text{goto label} \\
\text{beq} & \quad \text{if } s0 == s1 \quad \text{goto label} \\
\text{bgez} & \quad \text{if } s0 \geq 0 \quad \text{goto label}
\end{align*}
\]

- **Jump instructions**: (unconditional goto):

\[
\begin{align*}
\text{j} & \quad \text{goto instruction at “label:”} \\
\text{jr} & \quad \text{goto instruction at Memory[$a0$]}
\end{align*}
\]

- We can build while-loops, for-loops, repeat-until loops, and if-then-else constructs from these...
if ($t0===$t1) then /*blockA*/ else /*blockB*/

beq $t0, $t1, blockA  # if equal goto A
j blockB              # ... else goto B

blockA:
... instructions of blockA ...
j exit

blockB:
... instructions of blockB ...

exit:
... next part of program ...
repeat-until

repeat ... until $t0 > $t1

... initialize $t0, e.g. to 0 ... 

loop:

... instructions of loop ...

add $t0, $t0, 1       # increment $t0
ble $t0, $t1, loop   # if <= $t1, loop

• Other loop structures (for-loops, while-loops, etc) can be constructed similarly
Jump Instructions

• Recall **J-Type** instructions have 6-bit opcode and 26-bit target address
  – in #instructions (words), so effectively $2^{28}$ bits
• Assembler converts very distant conditional branches into inverse-branch and jump, e.g.

  ```assembly
  beq $2, $3, very_far_label
  /* next instruction */
  ```

• ... is converted to:

  ```assembly
  bne $2, $3, L1;    # continue
  j very_far_label;  # branch far
  L1:
  /*next instruction */
  ```
Indirect Jumps

• Sometimes we need to jump (or branch) more than $2^{28}$ bytes – can use *indirect jump* via register
  
  \[
  \text{j}r \; \$t1 \quad \# \text{ transfer control to} \\
  \quad \# \text{ memory address in } \$t1
  \]

• Can also use to build a *jump table*

• e.g. suppose we want to branch to different locations depending on the value held in $a0$

  .data
  
  jtab: \text{.word} \ l1, \ l2, \ l3, \ l4, \ l5, \ l6
  .text

  main: \text{... instructions setting } \$a0, \text{ etc ...}

  \text{l}w \; \$t7, \text{jtab}(\$a0) \quad \# \text{ load address}

  \text{j}r \; \$t7 \quad \# \text{ jump}

  \ l1: \text{... instructions ... }

  \ l2: \text{... instructions ... }

  \ l3: \text{... instructions ... (and so on...)}
The Spim Simulator

- “$^{1/25}$th the performance at none of the cost”
- Simulates a MIPS-based machine with some basic virtual hardware (console)
- Installation
  1. From the Patterson & Hennesey textbook CD
  2. From the internet
     http://www.cs.wisc.edu/~larus/spim.html
- Versions for Windows, Mac and Linux
PC Spim

reset "machine", load asm programs, run them, etc

register state (incl status reg)

.text section: (program)

.data section and the stack

diagnostic messages
Using SPIM

- Combines an assembler, a simulator and BIOS
- Assembly language program prepared in your favourite way as a text file
- Label your first instruction as `main`, e.g.
  ```
  main:    add  $5, $3, $4  # comment
  ```
- Read program into SPIM which will assemble it and may indicate assembly errors (1 at a time!)
- Execute your program (e.g. hit F5)
- Results output to window which simulates console (or by inspection of registers)
- Let’s look at an example...
SPIM System Calls

• As you’ll have noticed, SPIM allows us to use special code sequences, e.g.
  li $a0, 10  # load argument $a0=10
  li $v0, 1  # call code to print integer
  syscall    # print $a0
  – will print out “10” on the console

• The syscall instruction does various things depending on the value of $v0
  – this is very similar to how things work in a modern PC or Mac BIOS, albeit somewhat simpler

• (We’ll see why these are called “system calls” later on in the course...)
## SPIM System Call Codes

<table>
<thead>
<tr>
<th>Procedure</th>
<th>code $v0</th>
<th>argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>print int</td>
<td>1</td>
<td>$a0 contains number</td>
</tr>
<tr>
<td>print float</td>
<td>2</td>
<td>$f12 contains number</td>
</tr>
<tr>
<td>print double</td>
<td>3</td>
<td>$f12 contains number</td>
</tr>
<tr>
<td>print string</td>
<td>4</td>
<td>$a0 address of string</td>
</tr>
<tr>
<td>read int</td>
<td>5</td>
<td>res returned in $v0</td>
</tr>
<tr>
<td>read float</td>
<td>6</td>
<td>res returned in $f0</td>
</tr>
<tr>
<td>read double</td>
<td>7</td>
<td>res returned in $f0</td>
</tr>
<tr>
<td>read string</td>
<td>8</td>
<td>$a0 buffer, $a1 length</td>
</tr>
<tr>
<td>exit program</td>
<td>10</td>
<td>/* none */</td>
</tr>
</tbody>
</table>
Example: Print numbers 1 to 10

.data
newln: .asciiz "\n"
.text
.globl main
main:
    li $s0, 1  # $s0 = loop counter
    li $s1, 10 # $s1 = upper bound of loop
loop:
    move $a0, $s0  # print loop counter $s0
    li $v0, 1
    syscall
    li $v0, 4  # syscall for print string
    la $a0, newln
    syscall
    addi $s0, $s0, 1  # increase counter by 1
    ble $s0, $s1, loop  # if ($s0<=$s1) goto to loop
    li $v0, 10
    syscall  # exit
Example: Increase array elems by 5

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

- Long assembly programs get very unwieldy!

- **Procedures** or **subroutines** (similar to **methods** or **functions**) allow us to structure programs.

- Makes use of a new J-type instruction, **jal**:
  - `jal addr`  # jump-and-link
    - stores (current address + 4) into register $ra
    - jumps to address addr

- **jr $ra**
  - we’ve seen this before – an indirect jump
  - after a jal, this will return back to the main code
Example Using Procedures

.data
newline:.asciiz "\n"
.text
print_eol:
    li $v0, 4
    la $a0, newline
    syscall
    jr $ra
# procedure to print "\n"

print_int:
    li $v0, 1
    syscall
    jr $ra
# prints integer in $a0

main:
    li $s0, 1
    li $s1, 10
    move $a0, $s0
# print loop counter
    jal print_int
    jal print_eol
    addi $s0, $s0, 1
# increment loop counter
    ble $s0, $s1, loop
# continue unless $s0>$s1
Non-leaf Procedures

• Procedures are great, but what if have procedures invoking procedures?

```
procA: ... instructions to do stuff procA does ...
li $a0, 25  # prep to call procB
jal procB   # $ra = next address
jr $ra      # return to caller

procB: ... instructions to do stuff procB does ...
jr $ra      # return to caller
```

```
main:
li $a0, 10   # prep to call procA
jal procA    # $ra = next address

... rest of program ...
```

INFINITE LOOP!
The Stack

- Problem was that there’s only one $ra!
  - generally need to worry about other regs too
- We can solve this by saving the contents of registers in memory before doing procedure
  - Restore values from memory before return
- **The stack** is a way of organizing data in memory which is ideally suited for this purpose
  - Has so-called last-in-first-out (LIFO) semantics
  - *push* items onto the stack, *pop* items back off
- Think of a pile of paper on a desk
  - “pushing” an item is adding a piece of paper
  - “popping” is removing it
  - size of pile grows and shrinks over time
The Stack in Practice

- Register $sp holds address of top of stack
  - In SPIM this is initialized to 0x7FFF.EFFC
- A “push” stores data, and decrements $sp
- A “pop” reads back data, and increments $sp

```assembly
# $a0 holds 0xFEE
# ‘push’ $a0
sub $sp, $sp, 4
sw $a0, 0($sp)

# ‘pop’ $a0
lw $a0, 0($sp)
add $sp, $sp, 4
```

- We use the stack for parameter passing, storing return addresses, and saving and restoring other registers
Fibonacci... in assembly!

Fibonacci sequence:

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

0, 1, 1, 2, 3, 5, 8, 13, 21,...

Assembly code:

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

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

fib:
sub $sp,$sp,12           # save registers on stack
sw $a0, 0($sp)           # save $a0 = n
sw $s0, 4($sp)           # save $s0
sw $ra, 8($sp)           # save return address $ra
bgt $a0,1, gen           # if n>1 then goto generic case
move $v0,$a0             # output = input if n=0 or n=1
j rreg                   # goto restore registers

gen:
sub $a0,$a0,1            # param = n-1
jal fib                  # compute fib(n-1)
move $s0,$v0              # save fib(n-1)
sub $a0,$a0,1             # set param to n-2
jal fib                  # and make recursive call
add $v0, $v0, $s0         # $v0 = fib(n-2)+fib(n-1)

rreg:
lw  $a0, 0($sp)           # restore registers from stack
lw  $s0, 4($sp)           #
lw  $ra, 8($sp)           #
add $sp, $sp, 12         # decrease the stack size
jr $ra
Optional Assembly Ticks

- **Tick 0**: download SPIM (some version) and assemble + run the hello world program
- **Tick 2**: write an assembly program which reads in any 10 values from the keyboard, and prints them out lowest to highest
- **Tick 3 (**hard**)**: write an optimized version of the Fibonacci code presented here. You may wish do custom stack frame management for the base cases, and investigate tail-recursion.
  - see what Fibonacci number you can compute in 5 minutes with the original and optimized versions