Agenda for Today & Next Few Lectures

- LC-3 and MIPS Instruction Set Architectures
- LC-3 and MIPS assembly and programming
- Introduction to microarchitecture and single-cycle microarchitecture
- Multi-cycle microarchitecture
**Required Readings**

- **This week**
  - Von Neumann Model, LC-3, and MIPS
    - P&P, Chapters 4, 5
    - H&H, Chapter 6
    - P&P, Appendices A and C (ISA and microarchitecture of LC-3)
    - H&H, Appendix B (MIPS instructions)
  - Programming
    - P&P, Chapter 6
  - **Recommended:** H&H Chapter 5, especially 5.1, 5.2, 5.4, 5.5

- **Next week**
  - Introduction to microarchitecture and single-cycle microarchitecture
    - H&H, Chapter 7.1-7.3
    - P&P, Appendices A and C
  - Multi-cycle microarchitecture
    - H&H, Chapter 7.4
    - P&P, Appendices A and C
What Will We Learn Today?

- Assembly Programming
  - Programming constructs
  - Debugging
  - Conditional statements and loops in MIPS assembly
  - Arrays in MIPS assembly
  - Function calls
    - The stack
Recall: The Von Neumann Model

INPUT
Keyboard, Mouse, Disk…

OUTPUT
Monitor, Printer, Disk…

CONTROL UNIT
PC or IP Inst Register

MEMORY
Mem Addr Reg Mem Data Reg

PROCESSING UNIT
ALU TEMP
Recall: LC-3: A Von Neumann Machine

Figure 4.3 The LC-3 as an example of the von Neumann model
Recall: The Instruction Cycle

- FETCH
- DECODE
- EVALUATE ADDRESS
- FETCH OPERANDS
- EXECUTE
- STORE RESULT
Recall: The Instruction Set Architecture

- The ISA is the interface between what the software commands and what the hardware carries out.

- The ISA specifies:
  - The memory organization:
    - Address space (LC-3: $2^{16}$, MIPS: $2^{32}$)
    - Addressability (LC-3: 16 bits, MIPS: 32 bits)
    - Word- or Byte-addressable
  - The register set:
    - R0 to R7 in LC-3
    - 32 registers in MIPS
  - The instruction set:
    - Opcodes
    - Data types
    - Addressing modes
Our First LC-3 Program:
Use of Conditional Branches for Looping
An Algorithm for Adding Integers

- We want to write a program that adds 12 integers
  - They are stored in addresses 0x3100 to 0x310B
  - Let us take a look at the flowchart of the algorithm

![Flowchart](image-url)

**Variables:
- R1: initial address of integers
- R3: final result of addition
- R2: number of integers left to be added

- Check if R2 becomes 0 (done with all integers?)
  - Load integer in R4
  - Accumulate integer value in R3
  - Increment address R1
  - Decrement R2
A Program for Adding Integers in LC-3

- We use **conditional branch instructions** to create a loop

<table>
<thead>
<tr>
<th>Address</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3000</td>
<td>LEA</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0x00FF</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>x3001</td>
<td>AND</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x3002</td>
<td>AND</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x3003</td>
<td>ADD</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>x3004</td>
<td>BR</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>x3005</td>
<td>LDR</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x3006</td>
<td>ADD</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x3007</td>
<td>ADD</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>x3008</td>
<td>ADD</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>x3009</td>
<td>BR</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>n</td>
<td>z</td>
<td>p</td>
<td>-6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Address**

- **LEA** loads an address into a register.
- **ADD** adds two registers.
- **LDR** loads a memory address into a register.
- **BR** branches to a new instruction.

**Instructions**

- **R1 = PC + 0x00FF = 3100** // load address
- **R2 = R2 + 12** // initialize counter
- **R4 = M[R1 + 0]** // load value
- **R3 = R3 + R4** // accumulate
- **R4 = M[R1 + 1]** // load next value
- **R3 = R3 + R4** // accumulate
- **R2 = R2 + 1** // increment counter
- **BRnzp (PC + 6) = BRnzp 0x3004** // jump

**Flowchart**

1. **R1 ← x3100**
2. **R3 ← 0**
3. **R2 ← 12**
4. **R2 ?= 0**
5. **Yes**
   - **R4 ← M[R1]**
   - **R3 ← R3 + R4**
   - **Increment R1**
   - **Decrement R2**
6. **No**

**Instructions**

- **BRnzp (PC ‡ + 6) = BRnzp 0x3004** // jump

**Notes**

- **‡ This is the incremented PC**
- **Bit 5 to differentiate both ADD instructions**
The LC-3 Data Path Revisited
The LC-3 Data Path

We highlight some data path components used in the execution of the instructions in the previous slides (not shown in the simplified data path).
(Assembly) Programming
Programming Constructs

- Programming requires dividing a task, i.e., a unit of work into smaller units of work.
- The goal is to replace the units of work with programming constructs that represent that part of the task.
- There are three basic programming constructs:
  - Sequential construct
  - Conditional construct
  - Iterative construct
The sequential construct is used if the designated task can be broken down into two subtasks, one following the other.
Conditional Construct

- The conditional construct is used if the designated task consists of **doing one of two subtasks**, but not both.

  - Either subtask may be "do nothing"
  - After the correct subtask is completed, the program moves onward

- E.g., if-else statement, switch-case statement
Iterative Construct

- The iterative construct is used if the designated task consists of **doing a subtask a number of times**, but only **as long as some condition is true**.

  - For loop, while loop, do-while loop

  - Is the condition still “true”?

- E.g., for loop, while loop, do-while loop
Let us see how to use the **programming constructs in an example program**

- The example program **counts the number of occurrences of a character** in a text file
- It uses **sequential, conditional, and iterative constructs**
- We see how to write **conditional and iterative constructs with conditional branches**
Counting Occurrences of a Character

- We want to **write a program** that counts the occurrences of a **character** in a file
  - Character from the keyboard (TRAP instr.)
  - The file finishes with the character **EOT** (End Of Text)
    - That is called a **sentinel**
    - In this example, EOT = 4
  - Result to the monitor (TRAP instr.)

![Diagram of an algorithm to count occurrences of a character](image)

**R2**: counter  
**R3**: initial address  
**Input char**
**Read char from file**

Check if end of file  
Is it the searched char?  
Increment R2  
Increment address  
Move output to R0  
Output counter  
Halt the program

---

Programming constructs

- **Sequential**: Do first part to completion  
  - Do second part to completion  
- **Conditional**
  - True  
  - False
  - Test cond.
  - Subtask 1
  - Subtask 2
- **Iterative**
  - Test cond.
  - False
  - Subtask
  - True
  - Subtask
TRAP Instruction

- TRAP invokes an OS service call

LC-3 assembly

```assembly
TRAP 0x23;
```

<table>
<thead>
<tr>
<th>Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP</td>
</tr>
</tbody>
</table>

- **OP = 1111**
- **trapvect8 = service call**
  - **0x23 = Input a character from the keyboard**
  - **0x21 = Output a character to the monitor**
  - **0x25 = Halt the program**
We use **conditional branch instructions** to create a **loops** and **if statements**.

### Figure 5.17

A machine language program that implements the algorithm of Figure 5.16 on the monitor, it is necessary to first convert it to an ASCII code. Since we have assumed the count is less than 10, we can do this by putting a 0 in front of the 4-bit binary representation of the count. Note in Figure 5.16 the relationship between the binary value of each decimal digit between 0 and 9 and its corresponding ASCII code. Finally, the count is output to the monitor, and the program terminates.

Figure 5.17 is a machine language program that implements the flowchart to Figure 5.16. First the initialization steps. The instruction at x3000 clears R2 by ANDing it with x0000; the instruction at x3012 loads the value stored in x3012 into R3. This is the address of the first character in the file that is to be examined for occurrences of our character. Again, we note that this file can be anywhere in memory. Prior to starting execution at x3000, some sequence of instructions must have stored the first address of this file in x3012. Location x3002 contains the TRAP instruction, which requests the operating system to perform a service call on behalf of this program. The function requested, as identified by the 8-bit trapvector 00100011 (or, x23), is to input a character from the keyboard and load it into R0. Table A.2 lists trapvectors for all operating system service calls that can be performed on behalf of a user program. Note (from Table A.2) that x23 directs the operating system to perform the service call that reads the next character struck and loads it into R0. The instruction at x3003 loads the character pointed to by R3 into R1.

Then the process of examining characters begins. We start (x3004) by subtracting 4 (the ASCII code for EOT) from R1, and storing it in R4. If the result R2 = 0 // initialize counter

R3 = 0x3012 // initial address

TRAP 0x23 // input char to R0

R1 = M[R3] // char from file

R4 = R1 – 4 // char – EOT

BRz 0x300E // check if end of file

R1 = NOT(R1) // subtract char from file from input char for comparison

R1 = R1 + 1

R1 = R1 + R0

BRnp 0x300B

R2 = R2 + 1 // increment the counter

R3 = R3 + 1 // increment address

R1 = M[R3] // char from file

BRnzp 0x3004

R0 = M[0x3013] // output counter to monitor with TRAP

R0 = R0 + R2

TRAP 0x21

TRAP 0x25

Starting address of file

ASCII TEMPLATE
Let us do some reverse engineering to identify conditional constructs and iterative constructs.

### Programming Constructs in LC-3

<table>
<thead>
<tr>
<th>Address</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3000</td>
<td>AND</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x3001</td>
<td>LD</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x3002</td>
<td>TRAP</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>x3003</td>
<td>LDR</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x3004</td>
<td>ADD</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x3005</td>
<td>BR</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x3006</td>
<td>NOT</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>x3007</td>
<td>ADD</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x3008</td>
<td>ADD</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x3009</td>
<td>BR</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x300A</td>
<td>ADD</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x300B</td>
<td>ADD</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x300C</td>
<td>LDR</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x300D</td>
<td>BR</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x300E</td>
<td>LD</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x300F</td>
<td>ADD</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>x3010</td>
<td>TRAP</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>x3011</td>
<td>AND</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>x3012</td>
<td></td>
<td></td>
<td></td>
<td>Start</td>
<td>ing</td>
<td>address</td>
<td>of</td>
<td>file</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x3013</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 5.17**

A machine language program that implements the algorithm of Figure 5.16 on the monitor, it is necessary to first convert it to an ASCII code. Since we have assumed the count is less than 10, we can do this by putting a d i n front of the 4-bit binary representation of the count. Note in Figure E. the relationship between the binary value of each decimal digit between 0 and 9 and its corresponding ASCII code. Finally, the count is output to the monitor, and the program terminates.

**Figure 5.17** is a machine language program that implements the flow chart of Figure 5.16. First the initialization steps. The instruction at x3000 clears R2 by ANDing it with x0000; the instruction at x3001 loads the value stored in x3012 into R3. This is the address of the first character in the file that is to be examined for occurrences of our character. Again, we note that this file can be anywhere in memory. Prior to starting execution at x3000, some sequence of instructions must have stored the first address of this file in x3012. Location x3002 contains the TRAP instruction, which requests the operating system to perform a service call on behalf of this program. The function requested, as identified by the 8-bit trap vector 00100011 (or, x23), is to input a character from the keyboard and load it into R0. Table A.2 lists trap vectors for all operating system service calls that can be performed on behalf of a user program. Note (from Table A.2) that x23 directs the operating system to perform the service call that reads the next character struck and loads it into R0. The instruction at x3003 loads the character pointed to by R3 into R1.

Then the process of examining characters begins. We start (x3004) by subtracting 4 (the ASCII code for EOT) from R1, and storing it in R4. If the result

\[ R4 = R1 - 4 \] // char – EOT

BRz 0x300E // check if end of file

\[ R1 = \text{NOT}(R1) \] // subtract char from file from input char for comparison

R1 = R1 + 1

R1 = R1 + R0

BRnp 0x300B

R2 = R2 + 1 // increment the counter

BRnzp 0x3004

if (R1 == R0) {
  ...
  // increment the counter
}

```cpp
// do {
  ...
} while(R1 != EOT);
```
Debugging
Debugging

- Debugging is the process of removing errors in programs

- It consists of tracing the program, i.e., keeping track of the sequence of instructions that have been executed and the results produced by each instruction

- A useful technique is to partition the program into parts, often referred to as modules, and examine the results computed in each module

- High-level language (e.g., C programming language) debuggers: dbx, gdb, Visual Studio debugger

- Machine code debugging: Elementary interactive debugging operations
Interactive Debugging

When debugging interactively, it is important to be able to

1. Deposit values in memory and in registers, in order to test the execution of a part of a program in isolation

2. Execute instruction sequences in a program by using
   - RUN command: execute until HALT instruction or a breakpoint
   - STEP command: execute a fixed number of instructions

3. Stop execution when desired
   - SET BREAKPOINT command: stop execution at a specific instruction in a program

4. Examine what is in memory and registers at any point in the program
Example: Multiplying in LC-3 (Buggy)

- A program is necessary to multiply, since LC-3 does not have multiply instruction
  - The following program multiplies R4 and R5
  - Initially, R4 = 10 and R5 = 3
  - The program produces 40. What went wrong?
  - It is useful to annotate each instruction

<table>
<thead>
<tr>
<th>Address</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3200</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x3201</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>x3202</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>x3203</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x3204</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND 1 0 1</td>
<td>R2 = 0 // initialize register</td>
</tr>
<tr>
<td>ADD 0 0 1</td>
<td>R2 = R2 + R4</td>
</tr>
<tr>
<td>ADD 0 0 1</td>
<td>R5 = R5 - 1</td>
</tr>
<tr>
<td>BR 0 0 0</td>
<td>BRzp 0x3201</td>
</tr>
<tr>
<td>HALT 1 1 1</td>
<td>HALT // end program</td>
</tr>
</tbody>
</table>

The program produces 40, but it is expected to produce 30. The error lies in the addition instruction. The correct program should be:

```
AND 1 0 1
ADD 0 1 0
ADD 0 1 0
ADD 1 0 1
ADD 0 0 1
BR 0 0 0
HALT 1 1 1
```
Debugging the Multiply Program

We examine the contents of all registers after the execution of each instruction.

<table>
<thead>
<tr>
<th>Address</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3200</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x3201</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x3202</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x3203</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x3204</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R2 = 0 // initialize register</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R2 = R2 + R4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R5 = R5 – 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRzp 0x3201</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HALT // end program</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The branch condition codes were set wrong. The conditional branch should only be taken if R5 is positive.

Correct instruction:

```
BRp #–3 // BRp 0x3201
```
Easier Debugging with Breakpoints

- We could use a **breakpoint** to save some work
- Setting a breakpoint in 0x3203 (BR) allows us to examine the **results of each iteration of the loop**

<table>
<thead>
<tr>
<th>Address</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3200</td>
<td>AND</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x3201</td>
<td>ADD</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x3202</td>
<td>ADD</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x3203</td>
<td>BR</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>x3204</td>
<td>HALT</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

---

A good test should also consider the **corner cases**, i.e., unusual values that the programmer might fail to consider

One last question: Does this program work if the initial value of R5 is 0?
Conditional Statements
and Loops in MIPS Assembly
In MIPS, we create **conditional constructs with conditional branches** (e.g., beq, bne...)

**If Statement**

- **High-level code**
  
  ```
  if (i == j)  
    f = g + h;  
  f = f - i;
  ```

- **MIPS assembly**
  
  ```
  # $s0 = f, $s1 = g  
  # $s2 = h  
  # $s3 = i, $s4 = j
  bne $s3, $s4, L1
  add $s0, $s1, $s2
  L1: sub $s0, $s0, $s3
  ```

**Branch not equal**

Compares two values ($s3=i, s4=j) and jumps if they are different
We use the unconditional branch (i.e., j) to skip the “else” subtask if the “if” subtask is the correct one.

High-level code:

```plaintext
if (i == j)
    f = g + h;
else
    f = f - i;
```

1. Compare two values ($s3=i$, $s4=j$) and, if they are different, jump to L1, to execute the “else” subtask.

MIPS assembly:

```assembly
# $s0 = f, $s1 = g,
# $s2 = h
# $s3 = i, $s4 = j

bne $s3, $s4, L1
add $s0, $s1, $s2
j done

L1: sub $s0, $s0, $s3
done:
```

2. Jump to done, after executing the “if” subtask.
While Loop

- As in LC-3, the **conditional branch** (i.e., beq) checks the condition and the **unconditional branch** (i.e., j) jumps to the beginning of the loop.

**High-level code**

```c
// determines the power of 2 equal to 128
int pow = 1;
int x   = 0;

while (pow != 128) {
    pow = pow * 2;
    x = x + 1;
}
```

**MIPS assembly**

```mips
# $s0 = pow, $s1 = x
addi $s0, $0, 1
add  $s1, $0, $0
addi $t0, $0, 128
while: beq $s0, $t0, done
sll $s0, $s0, 1
addi $s1, $s1, 1
j    while
done:
```

1. Conditional branch to check if the condition still holds
2. Unconditional branch to the beginning of the loop
The implementation of the "for" loop is similar to the "while" loop.

High-level code:

```c
// add the numbers from 0 to 9
int sum = 0;
int i;
for (i = 0; i != 10; i = i+1) {
    sum = sum + i;
}
```

MIPS assembly:

```assembly
# $s0 = i, $s1 = sum
  addi $s1, $0, 0
  add $s0, $0, $0
  addi $t0, $0, 10
for:  beq $s0, $t0, done
    add $s1, $s1, $s0
    addi $s0, $s0, 1
    j for

done:  
```

1. Conditional branch to check if the condition still holds
2. Unconditional branch to the beginning of the loop
We use `slt` (i.e., set less than) for the "less than" comparison.

### High-level code

```c
// add the powers of 2 from 1 to 100
int sum = 0;
int i;
for (i = 1; i < 101; i = i*2) {
    sum = sum + i;
}
```

### MIPS assembly

```
# $s0 = i, $s1 = sum
addi $s1, $0, 0
addi $s0, $0, 1
addi $t0, $0, 101
loop:
    slt $t1, $s0, $t0
    beq $t1, $0, done
    add $s1, $s1, $s0
    sll $s0, $s0, 1
    j loop
done:
```

**Set less than**

\[ $t1 = \text{if } s0 < t0 \text{ then } 1 \text{ else } 0 \]

**Shift left logical**

\[ s0 = \text{left shift logical of } s0 \]
Arrays in MIPS
Arrays

- Accessing an array requires **loading the base address into a register**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x12340010</td>
<td>array[4]</td>
</tr>
<tr>
<td>0x1234800C</td>
<td>array[3]</td>
</tr>
<tr>
<td>0x12348008</td>
<td>array[2]</td>
</tr>
<tr>
<td>0x12348004</td>
<td>array[1]</td>
</tr>
<tr>
<td>0x12348000</td>
<td>array[0]</td>
</tr>
</tbody>
</table>

- In MIPS, this is something we **cannot do with one single immediate operation**

- **Load upper immediate + OR immediate**

  ```
  lui $s0, 0x1234
  ori $s0, $s0, 0x8000
  ```
We first load the **base address of the array** into a register (e.g., $s0) using **lui** and **ori**

### High-level code

```c
int array[5];

array[0] = array[0] * 2;

```

### MIPS assembly

```assembly
# array base address = $s0
# Initialize $s0 to 0x12348000
lui $s0, 0x1234
ori $s0, $s0, 0x8000

lw $t1, 0($s0)
sll $t1, $t1, 1
sw $t1, 0($s0)
lw $t1, 4($s0)
sll $t1, $t1, 1
sw $t1, 4($s0)
```
Function Calls
Function Calls

- **Why functions (i.e., procedures)?**
  - Frequently accessed code
  - Make a program more modular and readable

- Functions have **arguments** and **return value**

- **Caller**: calling function
  - main()

- **Callee**: called function
  - sum()

```c
void main()
{
    int y;
    y = sum(42, 7);
    ...
}

int sum(int a, int b)
{
    return (a + b);
}
```
Function Calls: Conventions

- Conventions

  - **Caller**
    - passes arguments
    - jumps to callee

  - **Callee**
    - performs the procedure
    - returns the result to caller
    - returns to the point of call
    - must not overwrite registers or memory needed by the caller
Function Calls in MIPS and LC-3

- Conventions in MIPS and LC-3

  - **Call procedure**
    - MIPS: Jump and link (jal)
    - LC-3: Jump to Subroutine (JSR, JSRR)

  - **Return from procedure**
    - MIPS: Jump register (jr)
    - LC-3: Return from Subroutine (RET)

  - **Argument values**
    - MIPS: $a0 - $a3

  - **Return value**
    - MIPS: $v0
## Function Calls: Simple Example

### High-level code

```c
int main() {
    simple();
    a = b + c;
}

void simple() {
    return;
}
```

### MIPS assembly

```
0x00400200 main: jal simple
0x00400204 add $s0,$s1,$s2

...  
0x00401020 simple: jr $ra
```

- **jal** jumps to `simple()` and saves PC+4 in the **return address register** ($ra)
  - $ra = 0x00400204

- In LC-3, **JSR(R)** put the return address in **R7**

- **jr $ra** jumps to address in $ra (LC-3 uses **RET** instruction)
Function Calls: Code Example

High-level code

```c
int main()
{
    int y;
    ...
    // 4 arguments
    y = differofsums(2, 3, 4, 5);
    ...
}

int differofsums(int f, int g, int h, int i)
{
    int result;
    result = (f + g) - (h + i);
    // return value
    return result;
}
```

MIPS assembly

```mips
# $s0 = y
main:
    ...
    addi $a0, $0, 2  # argument 0 = 2
    addi $a1, $0, 3  # argument 1 = 3
    addi $a2, $0, 4  # argument 2 = 4
    addi $a3, $0, 5  # argument 3 = 5
    jal differofsums  # call procedure
    add $s0, $v0, $0  # y = returned value
    ...
# $s0 = result
differofsums:
    add $t0, $a0, $a1  # $t0 = f + g
    add $t1, $a2, $a3  # $t1 = h + i
    sub $s0, $t0, $t1  # result = (f + g) - (h + i)
    add $v0, $s0, $0  # put return value in $v0
    jr $ra           # return to caller
```
What if the main function was using some of those registers?
- $t0, t1, s0$
- They could be overwritten by the function
- We can use the stack to temporarily store registers

MIPS assembly

```mips
diffofsums:
  add $t0, $a0, $a1  # $t0 = f + g
  add $t1, $a2, $a3  # $t1 = h + i
  sub $s0, $t0, $t1  # result=(f + g) - (h + i)
  add $v0, $s0, $0   # put return value in $v0
  jr $ra             # return to caller
```
The Stack

- The stack is a memory area used to save local variables

- It is a Last-In-First-Out (LIFO) queue

- The stack pointer ($sp) points to the top of the stack
  - It grows down in MIPS

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>7FFFFFFFC</td>
<td>12345678</td>
<td>7FFFFFFFC</td>
<td>12345678</td>
</tr>
<tr>
<td>7FFFFFFF8</td>
<td></td>
<td>7FFFFFFF8</td>
<td>AABBCCDD</td>
</tr>
<tr>
<td>7FFFFFFF4</td>
<td></td>
<td>7FFFFFFF4</td>
<td>11223344</td>
</tr>
<tr>
<td>7FFFFFF0</td>
<td></td>
<td>7FFFFFF0</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

Two words pushed on the stack
Saving and restoring all registers requires a lot of effort.

In MIPS, there is a convention about temporary registers (i.e., $t0-$t9): There is no need to save them.
- Programmers can use them for temporary/partial results.
Temporary registers $t0$-$t9$ are **nonpreserved** registers. They are not saved, thus, they can be overwritten by the function.

- Registers $s0$-$s7$ are **preserved** (saved; callee-saved) registers.
Lecture Summary

- Assembly Programming
  - Programming constructs
  - Debugging
  - Conditional statements and loops in MIPS assembly
  - Arrays in MIPS assembly
  - Function calls
    - The stack