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…

**MEMORY**
- Mem Addr Reg
- Mem Data Reg

**PROCESSING UNIT**
- ALU
- TEMP

**CONTROL UNIT**
- PC or IP
- Inst Register
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

<table>
<thead>
<tr>
<th>Problem</th>
<th>Algorithm</th>
<th>Program</th>
<th>ISA</th>
<th>Microarchitecture</th>
<th>Circuits</th>
<th>Electrons</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>ISA</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

**Figure 5.12** An algorithm for adding 12 integers

A flowchart for an algorithm to solve the problem is shown in Figure 5.12.

First, as in all algorithms, we must initialize our variables. That is, we must set up the initial values of the variables that the computer will use in executing the program that solves the problem. There are three such variables: the address of the next integer to be added (assigned to R1), the running sum (assigned to R3), and the number of integers left to be added (assigned to R2). The three variables are initialized as follows: The address of the first integer to be added is put in R1. R3, which will keep track of the running sum, is initialized to 0. R2, which will keep track of the number of integers left to be added, is initialized to 12. Then the process of adding begins.

The program repeats the process of loading into R4 one of the 12 integers, and adding it to R3. Each time we perform the ADD, we increment R1 so twill point to (i.e., contain the address of) the next number to be added and decrement R2 so we will know how many numbers still need to be added. When R2 becomes zero, the Z condition code is set, and we can detect that we are done.

The 10-instruction program shown in Figure 5.13 accomplishes this task.

The details of the program execution are as follows: The program starts with PC = x3000. The first instruction (at location x3000) loads R1 with the address x3100. (The incremented PC is x3001; the sign-extended PC offset is 00 FF.)

The instruction at x3001 clears R3. R3 will keep track of the running sum, so it must start off with the value 0. As we said previously, this is called initializing the SUM to zero.

The instructions at x3002 and x3003 set the value of R2 to 12, the number of integers to be added. R2 will keep track of how many numbers have already been added. This will be done (by the instruction contained in x3008) by decrementing R2 after each addition takes place.

The instruction at x3004 is a conditional branch instruction. Note that bit[10] is a 1. That means that the Z condition code will be examined. If it is set, we...
A Program for Adding Integers in LC-3

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

### Address 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

<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 1 1 0 0 0 1 0x00FF 1 1 1 1 1 1</td>
<td>R1 = PC + 0x00FF = 3100 // load address</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>x3001</td>
<td>AND 1 0 1 0 1 1 0 1 1 1 0 0 0 0 0</td>
<td>R3 = 0 // reset 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>
</tr>
<tr>
<td>x3002</td>
<td>AND 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0</td>
<td>R2 = 0 // reset 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>
</tr>
<tr>
<td>x3003</td>
<td>ADD 0 0 1 0 1 0 0 1 0 1 0 1 1 1 0 0</td>
<td>R2 = R2 + 12 // initialize counter</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>x3004</td>
<td>BR 0 0 0 0 z 0 0 0 0 0 0 0 1 0 1</td>
<td>BRz (PC + 5) = BRz 0x300A // check condition</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>x3005</td>
<td>LDR 1 1 0 1 0 0 0 0 1 0 0 0 0 0 0</td>
<td>R4 = M[R1 + 0] // load value</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>x3006</td>
<td>ADD 0 0 1 0 1 1 0 1 1 0 0 0 1 0 0</td>
<td>R3 = R3 + R4 // accumulate</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>x3007</td>
<td>ADD 0 0 1 0 0 1 0 0 1 1 1 0 0 0 1</td>
<td>R1 = R1 + 1 // increment address</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>x3008</td>
<td>ADD 0 0 1 0 1 0 0 1 0 1 1 1 1 1 1</td>
<td>R2 = R2 – 1 // decrement counter</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>x3009</td>
<td>BR 0 0 0 n z p 1 1 1 1 1 0 1 0</td>
<td>BRnzp (PC – 6) = BRnzp 0x3004 // jump</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>

**Flowchart**

1. **R1** ← x3100
2. **R3** ← 0
3. **R2** ← 12

4. Yes
5. **R2** ? = 0
6. **R4** ← M[R1]
7. **R3** ← R3 + R4
8. Increment **R1**
9. Decrement **R2**

10. Yes
11. **R2** ? = 0
12. **R4** ← M[R1]
13. **R3** ← R3 + R4
14. Increment **R1**
15. Decrement **R2**

**Bit 5 to differentiate both ADD instructions**

✝ This is the incremented PC
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
Sequential Construct

- The sequential construct is used if the designated task can be **broken down into two subtasks**, one following the other.
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.

- 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 will 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.)

Programming constructs

Sequential

- Do first part to completion
- Do second part to completion

Conditional

- Test cond. True Subtask 1 False Subtask 2

Iterative

- Test cond. True Subtask False Subtask

Figure 5.16 An algorithm to count occurrences of a character

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
TRAP Instruction

- TRAP invokes an OS service call

LC-3 assembly

```
TRAP 0x23;
```

- Machine Code

<table>
<thead>
<tr>
<th>OP</th>
<th>Trapvect8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td></td>
</tr>
</tbody>
</table>

- 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.

Counting Occurrences of a Char in LC-3

Address 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
x3000 \text{AND} \quad 0 1 \quad 0 1 0 1 0 1 0 0 0 0 0 0
x3001 \text{LD} \quad 0 1 0 \quad 0 1 1 0 0 0 0 1 0 0 0 0
x3002 \text{TRAP} \quad 1 1 0 0 0 0 0 0 1 0 0 0 1 1
x3003 \text{LDR} \quad 1 0 0 0 1 0 1 1 0 0 0 0 0 0
x3004 \text{ADD} \quad 0 1 1 0 0 0 1 1 1 1 1 0 0 0
x3005 \text{BR} \quad 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
x3006 \text{NOT} \quad 0 1 0 0 1 0 0 1 1 1 1 1 1 1 1
x3007 \text{ADD} \quad 0 1 0 0 1 0 0 1 1 0 0 0 0 1
x3008 \text{ADD} \quad 0 1 0 0 1 0 0 1 0 0 0 0 0 0
x3009 \text{BR} \quad 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
x300A \text{ADD} \quad 0 1 0 1 0 0 1 0 1 0 0 0 0 1
x300B \text{ADD} \quad 0 1 0 1 1 0 1 1 0 0 0 0 1
x300C \text{LDR} \quad 1 0 0 0 1 0 1 1 0 0 0 0 0 0
x300D \text{BR} \quad 0 0 0 0 0 0 0 1 1 1 1 0 1 1 0
x300E \text{LD} \quad 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0
x300F \text{ADD} \quad 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1
x3010 \text{TRAP} \quad 1 1 0 0 0 0 0 0 1 0 0 0 0 1
x3011 \text{AND} \quad 1 1 0 0 0 0 0 0 1 0 0 1 0 0 1
x3012 \text{ASCII TEMPLATE} \quad 0 0 0 0 0 1 1 0 0 0 0
x3013 Starting address of file

R2 = 0 // initialize counter
R3 = M[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)
R1 = R1 + 1
R1 = R1 + R0
// subtract char from file from input char for comparison
BRnp 0x300B
R2 = R2 + 1 // increment the counter
R3 = R3 + 1 // increment address
R1 = M[R3] // char from file
BRnzp 0x3004
R0 = M[0x3013]
R0 = R0 + R2
TRAP 0x21
TRAP 0x25
// output counter to monitor with TRAP

Figure 5.17 is a machine language program that implements the flow chart to 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 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

\begin{equation}
\text{R2} = 0 \quad // \text{initialize counter}
\end{equation}

\begin{equation}
\text{R3} = \text{M}[0x3012] \quad // \text{initial address}
\end{equation}

\begin{equation}
\text{TRAP} \ 0x23 \quad // \text{input char to R0}
\end{equation}

\begin{equation}
\text{R1} = \text{M}[R3] \quad // \text{char from file}
\end{equation}

\begin{equation}
\text{R4} = \text{R1} - 4 \quad // \text{char – EOT}
\end{equation}

\begin{equation}
\text{BRz} \ 0x300E \quad // \text{check if end of file}
\end{equation}

\begin{equation}
\text{R1} = \text{NOT}(R1)
\end{equation}

\begin{equation}
\text{R1} = \text{R1} + 1
\end{equation}

\begin{equation}
\text{R1} = \text{R1} + \text{R0}
\end{equation}

\begin{equation}
// \text{subtract char from file from input char for comparison}
\end{equation}

\begin{equation}
\text{BRnp} \ 0x300B
\end{equation}

\begin{equation}
\text{R2} = \text{R2} + 1 \quad // \text{increment the counter}
\end{equation}

\begin{equation}
\text{R3} = \text{R3} + 1 \quad // \text{increment address}
\end{equation}

\begin{equation}
\text{R1} = \text{M}[R3] \quad // \text{char from file}
\end{equation}

\begin{equation}
\text{BRnzp} \ 0x3004
\end{equation}

\begin{equation}
\text{R0} = \text{M}[0x3013]
\end{equation}

\begin{equation}
\text{R0} = \text{R0} + \text{R2}
\end{equation}

\begin{equation}
\text{TRAP} \ 0x21
\end{equation}

\begin{equation}
\text{TRAP} \ 0x25
\end{equation}

\begin{equation}
// \text{output counter to monitor with TRAP}
\end{equation}

Starting address of file

ASCII TEMPLATE

0 0 0 0 0 1 1 0 0 0 0

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 0011 in 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.
Programming Constructs in LC-3

- Let us do some reverse engineering to identify conditional constructs and iterative constructs

### Address | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0
---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---
x3000 | AND | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
x3001 | LD | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
x3002 | TRAP | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
x3003 | LDR | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
x3004 | ADD | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
x3005 | BR | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
x3006 | NOT | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
x3007 | ADD | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
x3008 | ADD | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
x3009 | BR | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
x300A | ADD | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
x300B | ADD | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
x300C | LDR | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
x300D | BR | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
x300E | LD | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
x300F | ADD | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
x3010 | TRAP | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
x3011 | AND | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
x3012 | | | | | | | | | | | | | | | | |
x3013 | | | | | | | | | | | | | | | | |

**Starting address of file**

```
while (R1 != EOT) {
    ...
}
```

```
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
```

```
if (R1 == R0) {
    ...
    // increment the counter
}
```
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 N command: execute a fixed number (N) 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

| Address | 15  | 14  | 13  | 12  | 11  | 10  | 9   | 8   | 7   | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|---------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| x3200   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| x3201   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| x3202   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| x3203   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| x3204   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

- **AND 1 0 1**
- **ADD 0 0 1**
- **ADD 0 0 1**
- **BR 0 0 0**
- **HALT 1 1 1**

- **R2 = 0 // initialize register**
- **R2 = R2 + R4**
- **R5 = R5 – 1**
- **BRzp 0x3201**
- **HALT // end program**
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>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>
<td>0</td>
</tr>
<tr>
<td>x3201</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>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>0</td>
<td>0</td>
<td>1</td>
<td>1</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>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>x3203</td>
<td>0</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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>x3204</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>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- **R2 = 0** // initialize register
- **R2 = R2 + R4**
- **R5 = R5 - 1**
- **BRzp 0x3201**
- **HALT // end program**

- **Correct result**: The branch condition codes were set wrong. The conditional branch should only be taken if R5 is positive.

- **Correct instruction**: `BRp #−3` // BRp 0x3201
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.

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...)

**High-level code**

```plaintext
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;
```

### MIPS 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:
```

1. Compare two values ($s3=i$, $s4=j$) and, if they are different, jump to L1, to execute the "else" subtask.
2. Jump to done, after executing the "if" subtask.
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
define the power
// of 2 equal to 128
int pow = 1;
int x   = 0;

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

**MIPS assembly**

```assembly
# $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
For Loop Using SLT

- 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

```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

$\texttt{slt} = \texttt{$s0 < $t0 ? 1:0}$

Shift left logical
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>0x1234000C</td>
<td>array[3]</td>
</tr>
<tr>
<td>0x12340008</td>
<td>array[2]</td>
</tr>
<tr>
<td>0x12340004</td>
<td>array[1]</td>
</tr>
<tr>
<td>0x12340000</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
  ```
Arrays: Code Example

- 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
We did not cover the following slides in lecture. These are for your preparation.
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 = diffofsums(2, 3, 4, 5);
    ...
}

int diffofsums(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 diffofsums    # call procedure
add $s0, $v0, $0  # y = returned value

# $s0 = result
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
```

Argument values
Return value
Return address
Function Calls: Need for the Stack

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
```

- 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
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>7FFFFFF8</td>
<td></td>
<td>7FFFFFF8</td>
<td>AABBCCDD</td>
</tr>
<tr>
<td>7FFFFFF4</td>
<td></td>
<td>7FFFFFF4</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>
<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.
MIPS Stack: Register Saving Convention

MIPS assembly

diffofsums:

```
addi $sp, $sp, -4       # allocate space on stack to store 1 register
sw  $s0, 0($sp)         # save $s0 on stack

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

lw  $s0, 0($sp)         # restore $s0 from stack
addi $sp, $sp, 4       # deallocate stack space
jr  $ra                 # return to caller
```

- 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
Digital Design & Computer Arch.
Lecture 10b: Assembly Programming

Prof. Onur Mutlu

ETH Zürich
Spring 2020
20 March 2020