Design of Digital Circuits

Lecture 10: ISA (II)

and Assembly Programming

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zurich
Spring 2018
23 March 2018
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
- Microprogramming
Readings

**This week**
- Von Neumann Model, LC-3, and MIPS
  - P&P, Chapter 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
- Digital Building Blocks
  - H&H, Chapter 5

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

- Instruction Set Architectures: LC-3 and MIPS
  - Operate instructions
  - Data movement instructions
  - Control instructions
- Instruction formats
- Addressing modes

- 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...

PROCESSING UNIT
ALU
TEMP

MEMORY
Mem Addr Reg
Mem Data Reg

CONTROL UNIT
PC or IP
Inst Register

OUTPUT
Monitor, Printer, Disk...
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
Operate Instructions
Operate Instructions

- In LC-3, there are three operate instructions
  - NOT is a **unary operation** (one source operand)
    - It executes bitwise NOT
  - ADD and AND are **binary operations** (two source operands)
    - ADD is 2’s complement addition
    - AND is bitwise SR1 & SR2

- In MIPS, there are many more
  - Most of R-type instructions (they are **binary operations**)
    - E.g., add, and, nor, xor...
  - I-type versions of the R-type operate instructions
  - F-type operations, i.e., floating-point operations
NOT in LC-3

- NOT assembly and machine code
  
  LC-3 assembly
  
  NOT R3, R5

Field Values

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>SR</th>
<th>Field Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>3</td>
<td>5</td>
<td>1 1 1 1 1 1</td>
</tr>
</tbody>
</table>

Machine Code

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>SR</th>
<th>Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 1</td>
<td>0 1 1</td>
<td>0 0 1</td>
<td>1 1 1 1 1 1</td>
</tr>
</tbody>
</table>

There is no NOT in MIPS. How is it implemented?
Operate Instructions

- We are already familiar with LC-3’s ADD and AND with register mode (R-type in MIPS)

- Now let us see the versions with one literal (i.e., immediate) operand

- Subtraction is another necessary operation
  - How is it implemented in LC-3 and MIPS?
Operate Instr. with one Literal in LC-3

- ADD and AND

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>SR1</th>
<th>1</th>
<th>imm5</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bits</td>
<td>3 bits</td>
<td>3 bits</td>
<td>5 bits</td>
<td></td>
</tr>
</tbody>
</table>

- OP = operation
  - E.g., ADD = 0001 (same OP as the register-mode ADD)
    - DR ← SR1 + sign-extend(imm5)
  - E.g., AND = 0101 (same OP as the register-mode AND)
    - DR ← SR1 AND sign-extend(imm5)

- SR1 = source register
- DR = destination register
- imm5 = Literal or immediate (sign-extend to 16 bits)
ADD with one Literal in LC-3

- ADD assembly and machine code

LC-3 assembly

ADD R1, R4, #−2

Field Values

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>SR</th>
<th>imm5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>-2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Machine Code

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>SR</th>
<th>imm5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>001</td>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>11110</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For example, if R4 contains the value 6 and R5 contains the value −18, then after the following instruction is executed

ADD R1, R4, #−2

R1 will contain the value −12.

If bit [5] is 1, the second source operand is contained within the instruction. In fact, the second source operand is obtained by sign-extend ing bits [4:0] to 16 bits before performing the ADD or AND. Figure 5.5 shows the key parts of the data path that are used to perform the instruction ADD R1, R4, #−2.

Since the immediate operand in an ADD or AND instruction must fit in bits [4:0] of the instruction, not all 2's complement integers can be used as immediate operands. Which integers are OK (i.e., which integers can be used as immediate operands)?

Figure 5.5 Data path relevant to the execution of ADD R1, R4, #−2
Instructions with one Literal in MIPS

- **I-type**
  - 2 register operands and immediate
- **Some operate and data movement instructions**

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- opcode = operation
- rs = source register
- rt =
  - destination register in some instructions (e.g., addi, lw)
  - source register in others (e.g., sw)
- imm = Literal or immediate
Add with one Literal in MIPS

- Add immediate

**MIPS assembly**

```
addi $s0, $s1, 5
```

<table>
<thead>
<tr>
<th>Field Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

```
rt ← rs + sign-extend(imm)
```

**Machine Code**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>001000</td>
<td>1001</td>
<td>1010</td>
<td>0000 0000 0000 0101</td>
</tr>
</tbody>
</table>

```
0x22300005
```
Subtract in LC-3

- **MIPS assembly**
  
  **High-level code**
  
  \[ a = b + c - d; \]
  
  **MIPS assembly**
  
  \[
  \begin{align*}
  \text{add} & \quad \$t0, \$s0, \$s1 \\
  \text{sub} & \quad \$s3, \$t0, \$s2
  \end{align*}
  \]

- **LC-3 assembly**
  
  **High-level code**
  
  \[ a = b + c - d; \]
  
  **LC-3 assembly**
  
  \[
  \begin{align*}
  \text{ADD} & \quad R2, R0, R1 \\
  \text{NOT} & \quad R4, R3 \\
  \text{ADD} & \quad R5, R4, #1 \\
  \text{ADD} & \quad R6, R2, R5
  \end{align*}
  \]

- **Tradeoff in LC-3**
  
  - More instructions
  - But, simpler control logic
Subtract Immediate

- **MIPS assembly**

  High-level code
  
  ```
  a = b - 3;
  ```

  MIPS assembly
  
  ```
  subi $s1, $s0, 3
  ```

  **Is subi necessary in MIPS?**

  MIPS assembly
  
  ```
  addi $s1, $s0, -3
  ```

- **LC-3**

  High-level code
  
  ```
  a = b - 3;
  ```

  LC-3 assembly
  
  ```
  ADD R1, R0, #–3
  ```
Data Movement Instructions and Addressing Modes
Data Movement Instructions

- In LC-3, there are seven data movement instructions
  - LD, LDR, LDI, LEA, ST, STR, STI

- Format of load and store instructions
  - Opcode (bits [15:12])
  - DR or SR (bits [11:9])
  - Address generation bits (bits [8:0])
  - Four ways to interpret bits, called addressing modes
    - PC-Relative Mode
    - Indirect Mode
    - Base+offset Mode
    - Immediate Mode

- In MIPS, there are only Base+offset and immediate modes for load and store instructions
PC-Relative Addressing Mode

- **LD (Load) and ST (Store)**

  - **OP** = opcode
    - E.g., **LD** = 0010
    - E.g., **ST** = 0011

  - **DR** = destination register in **LD**
  - **SR** = source register in **ST**

  - **LD**: \( DR \leftarrow \text{Memory}[\text{PC}^\dagger + \text{sign}-\text{extend}(\text{PCoffset9})] \)

  - **ST**: \( \text{Memory}[\text{PC}^\dagger + \text{sign}-\text{extend}(\text{PCoffset9})] \leftarrow SR \)

\( ^\dagger \text{This is the incremented PC} \)
LD in LC-3

- LD assembly and machine code

**LC-3 assembly**

LD R2, 0x1AF

**Field Values**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>PCoffset9</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>0x1AF</td>
</tr>
</tbody>
</table>

**Machine Code**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>PCoffset9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010</td>
<td>010</td>
<td>110101111</td>
</tr>
</tbody>
</table>

![Diagram of LD R2, 0x1AF execution]

**Limitation:** The PC-relative addressing mode cannot address far away from the instruction.

The memory address is only +256 to -255 locations away of the LD or ST instruction.
Indirect Addressing Mode

- LDI (Load Indirect) and STI (Store Indirect)

<table>
<thead>
<tr>
<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>OP</td>
<td></td>
<td></td>
<td></td>
<td>DR</td>
<td></td>
<td></td>
<td>SR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- OP = opcode
  - E.g., LDI = 1010
  - E.g., STI = 1011

- DR = destination register in LDI
- SR = source register in STI

- LDI: \( \text{DR} \leftarrow \text{Memory[Memory[PC}^\dagger + \text{sign-extend(PCoffset9)]]} \)

- STI: \( \text{Memory[Memory[PC}^\dagger + \text{sign-extend(PCoffset9)]]} \leftarrow \text{SR} \)

\( ^\dagger \text{This is the incremented PC} \)
LDI in LC-3

- LDI assembly and machine code

**LC-3 assembly**

LDI R3, 0x1CC

**Field Values**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>PCoffset9</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>0x1CC</td>
</tr>
</tbody>
</table>

**Machine Code**

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>PCoffset9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010</td>
<td>011</td>
<td>111001100</td>
</tr>
</tbody>
</table>

Now the address of the operand can be anywhere in the memory.
Base+Offset Addressing Mode

- LDR (Load Register) and STR (Store Register)

- **OP** = opcode
  - E.g., LDR = 0110
  - E.g., STR = 0111

- **DR** = destination register in LDR
- **SR** = source register in STR

- **LDR:** \( DR \leftarrow \text{Memory[BaseR + sign-extend(offset6)]} \)

- **STR:** \( \text{Memory[BaseR + sign-extend(offset6)]} \leftarrow SR \)
LDR in LC-3

- LDR assembly and machine code

LC-3 assembly

```
LDR R1, R2, 0x1D
```

Field Values

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>BaseR</th>
<th>offset6</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1</td>
<td>2</td>
<td>0x1D</td>
</tr>
</tbody>
</table>

Machine Code

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>BaseR</th>
<th>offset6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0110</td>
<td>001</td>
<td>010</td>
<td>011101</td>
</tr>
</tbody>
</table>

The address of the operand can also be anywhere in the memory.
In MIPS, lw and sw use base+offset mode (or base addressing mode)

High-level code


MIPS assembly

sw $s3, 8($s0)

Memory[$s0 + 8] ← $s3

Field Values

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
<td>16</td>
<td>19</td>
<td>8</td>
</tr>
</tbody>
</table>

imm is the 16-bit offset, which is sign-extended to 32 bits.
# An Example Program in MIPS and LC-3

<table>
<thead>
<tr>
<th>High-level code</th>
<th>MIPS registers</th>
<th>LC-3 registers</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a = A[0];</code></td>
<td><code>A = $s0</code></td>
<td><code>A = R0</code></td>
</tr>
<tr>
<td><code>c = a + b - 5;</code></td>
<td><code>b = $s2</code></td>
<td><code>b = R2</code></td>
</tr>
<tr>
<td><code>B[0] = c;</code></td>
<td><code>B = $s1</code></td>
<td><code>B = R1</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MIPS assembly</th>
<th>LC-3 assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lw  $t0, 0($s0)</code></td>
<td><code>LDR  R5, R0, #0</code></td>
</tr>
<tr>
<td><code>add  $t1, $t0, $s2</code></td>
<td><code>ADD  R6, R5, R2</code></td>
</tr>
<tr>
<td><code>addi $t2, $t1, -5</code></td>
<td><code>ADD  R7, R6, #-5</code></td>
</tr>
<tr>
<td><code>sw  $t2, 0($s1)</code></td>
<td><code>STR  R7, R1, #0</code></td>
</tr>
</tbody>
</table>
Immediate Addressing Mode

- LEA (Load Effective Address)
  - OP = 1110
  - DR = destination register
  - LEA: \( DR \leftarrow PC^\dagger + \text{sign-extend}(\text{PCoffset9}) \)

What is the difference from PC-Relative addressing mode?

Answer: Instructions with PC-Relative mode access memory, but LEA does not

\(^\dagger\)This is the incremented PC
LEA in LC-3

- LEA assembly and machine code

LC-3 assembly

LEA R5, #−3

Field Values

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>PCoffset9</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>5</td>
<td>0x1FD</td>
</tr>
</tbody>
</table>

Machine Code

<table>
<thead>
<tr>
<th>OP</th>
<th>DR</th>
<th>PCoffset9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110</td>
<td>101</td>
<td>111111101</td>
</tr>
</tbody>
</table>

Note that the Base+offset addressing mode also allows the address of the operand to be anywhere in the computer's memory.

5.3.4 Immediate Mode

The fourth and last addressing mode used by the data movement instructions is the immediate (or, literal) addressing mode. It is used only with the load effective address (LEA) instruction. LEA (opcode = 1110) loads the register specified by bits [11:9] of the instruction with the value formed by adding the incremented program counter to the sign-extended bits [8:0] of the instruction. The immediate addressing mode is so named because the operand to be loaded into the destination register is obtained immediately, that is, without requiring any access to memory.

The LEA instruction is useful to initialize a register with an address that is very close to the address of the instruction doing the initializing. If memory location x4018 contains the instruction LEA R5, #−3, and the PC contains x4018, R5 will contain x4016 after the instruction at x4018 is executed.

Figure 5.9 shows the relevant parts of the data path required to execute the LEA instruction. Note that no access to memory is required to obtain the value to be loaded.
Immediate Addressing Mode in MIPS

- In MIPS, `lui` (load upper immediate) loads a 16-bit immediate into the upper half of a register and sets the lower half to 0.

- It is used to assign 32-bit constants to a register.

<table>
<thead>
<tr>
<th>High-level code</th>
<th>MIPS assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a = 0x6d5e4f3c;</code></td>
<td><code># $s0 = a</code></td>
</tr>
<tr>
<td></td>
<td><code>lui $s0, 0x6d5e</code></td>
</tr>
<tr>
<td></td>
<td><code>ori $s0, 0x4f3c</code></td>
</tr>
</tbody>
</table>
What is the final value of R3?

Addressing Example 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>x30F6</td>
<td>1</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>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>x30F7</td>
<td>0</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>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>x30F8</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>0</td>
<td>1</td>
</tr>
<tr>
<td>x30F9</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>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>x30FA</td>
<td>0</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>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>x30FB</td>
<td>0</td>
<td>1</td>
<td>1</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>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>x30FC</td>
<td>1</td>
<td>0</td>
<td>1</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>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

R1 ← PC - 3
R2 ← R1 + 14
M[x30F4] ← R2
R2 ← 0
R2 ← R2 + 5
M[R1 + 14] ← R2
R3 ← M[M[x30F4]]
What is the final value of R3?

The final value of R3 is 5.
Control Flow Instructions
Control Flow Instructions

- Allow a program to execute out of sequence

- Conditional branches and jumps
  - Conditional branches are used to make decisions
    - E.g., if-else statement
  - In LC-3, three condition codes are used
  - Jumps are used to implement
    - Loops
    - Function calls
  - JMP in LC-3 and j in MIPS
Condition Codes in LC-3

- Each time one GPR (R0-R7) is written, three single-bit registers are updated.

- Each of these *condition codes* are either set (set to 1) or cleared (set to 0).
  - If the written value is **negative**
    - N is set, Z and P are cleared.
  - If the written value is **zero**
    - Z is set, N and P are cleared.
  - If the written value is **positive**
    - P is set, N and P are cleared.

- SPARC and x86 are examples of ISAs that use condition codes.
Conditional Branches in LC-3

- **BRz (Branch if Zero)**

<table>
<thead>
<tr>
<th>BRz</th>
<th>PCoffset9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>n</td>
</tr>
<tr>
<td>4 bits</td>
<td></td>
</tr>
</tbody>
</table>

- **n, z, p = which N, Z, and/or P is tested**

- **PCoffset9 = immediate or constant value**

- **if \( ((n \text{ AND } N) \text{ OR } (p \text{ AND } P) \text{ OR } (z \text{ AND } Z)) \)**
  - then \( \text{PC} \leftarrow \text{PC}^+ + \text{sign-extend(} \text{PCoffset9} \text{)} \)

- **Variations: BRn, BRz, BRp, BRzp, BRnp, BRnz, BRnzp**

\[^{†}\text{This is the incremented PC}\]
Conditional Branches in LC-3

- **BRz**

**BRz 0x0D9**

What if \( n = z = p = 1 \)? (i.e., **BRnzp**)

And what if \( n = z = p = 0 \)?

**Figure 5.11** Data path relevant to the execution of **BRz x0D9**
Conditional Branches in MIPS

- **beq (Branch if Equal)**

  ```
  beq $s0, $s1, offset
  ```

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>rs</th>
<th>rt</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- 4 = opcode
- rs, rt = source registers
- offset = immediate or constant value
- if rs == rt
  - then PC ← PC\(^\dagger\) + sign-extend(offset) \* 4
- Variations: beq, bne, blez, bgtz

\(^\dagger\)This is the incremented PC
Branch If Equal in MIPS and LC-3

- This is an example of tradeoff in the instruction set
  - The same functionality requires more instructions in LC-3
  - But, the control logic requires more complexity in MIPS
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

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 R1 will 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 x00FF.)

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

R1: initial address
R3: final result
R2: number of integers left to be added

Check if R2 becomes 0

Load integer in R4
Accumulate in R3
Increment address R1
Decrement R2
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>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>
</tr>
<tr>
<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></td>
</tr>
<tr>
<td></td>
<td>R1 = PC$^+$ + 0x00FF = 3100</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>x3001</td>
<td>AND</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</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>
<td>0</td>
</tr>
<tr>
<td></td>
<td>R3 = 0</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>x3002</td>
<td>AND</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></td>
<td>R2 = 0</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>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>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>R2 = R2 + 12</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>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>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>BRz (PC$^+$ + 5) = BRz 0x300A$^+$</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>x3005</td>
<td>LDR</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>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>R4 = M[R1 + 0]</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>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></td>
<td>R3 = R3 + 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>
</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>0</td>
</tr>
<tr>
<td></td>
<td>R1 = R1 + 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>
</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></td>
<td>R2 = R2 − 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>
</tr>
<tr>
<td>x3009</td>
<td>BR</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>n</td>
<td>z</td>
<td>p</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></td>
<td>BRnzp (PC$^+$ − 6) = BRnzp 0x3004</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>

$^+$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.
Constructs in an Example Program

- 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 program flow]
## TRAP Instruction

- TRAP invokes an **OS service call**

### LC-3 assembly

| TRAP 0x23; |

### Machine Code

<table>
<thead>
<tr>
<th>OP</th>
<th>0000</th>
<th>trapvect8</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 bits</td>
<td>8 bits</td>
<td></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
### Counting Occurrences of a Char in LC-3

- **We use conditional branch instructions to create a loops and if statements**

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

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

---

**Figure 5.17**: A machine language program that implements the algorithm 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 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

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
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
Conditional constructs and iterative constructs

Let us do some reverse engineering

Programming Constructs in LC-3

Address 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
x3000 0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0
x3001 0 0 1 0 0 1 1 0 0 0 0 1 0 0 0 0
x3002 1 1 1 1 0 0 0 0 0 0 1 0 0 0 1 1
x3003 0 1 1 0 0 0 1 0 1 1 0 0 0 0 0 0
x3004 0 0 0 1 1 0 0 0 0 1 1 1 1 1 0 0
x3005 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0
x3006 1 0 0 1 0 0 1 0 0 1 1 1 1 1 1 1
x3007 0 0 0 1 0 0 1 0 0 1 1 0 0 0 0 1
x3008 0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0
x3009 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0
x300A 0 0 0 1 0 1 0 0 1 0 1 0 0 0 1 1
x300B 0 0 0 1 0 1 1 0 1 1 1 0 0 0 0 1
x300C 0 1 1 0 0 0 1 0 1 1 0 0 0 0 0 0
x300D 0 0 0 0 1 1 1 1 1 1 1 0 1 1 0 0
x300E 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0
x300F 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0
x3010 1 1 1 1 0 0 0 0 0 0 1 0 0 0 0 1
x3011 1 1 1 1 0 0 0 0 0 0 1 0 0 1 0 1
x3012 Starting address of file
x3013 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0

do
{
...
}while(R1 != EOT);

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.

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

- 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>0</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>0</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>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x3202</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>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>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>
</tr>
<tr>
<td>x3204</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>
<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>
</tbody>
</table>

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>
</tr>
</thead>
<tbody>
<tr>
<td>x3200</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>x3201</td>
<td>0</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>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>x3202</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>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>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>
</tr>
<tr>
<td>x3204</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>
<td>0</td>
<td>1</td>
<td>0</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

We should only take the conditional branch if R5 is positive.

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**.

<table>
<thead>
<tr>
<th>PC</th>
<th>R2</th>
<th>R4</th>
<th>R5</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3203</td>
<td>10</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>x3203</td>
<td>20</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>x3203</td>
<td>30</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>x3203</td>
<td>40</td>
<td>10</td>
<td>-1</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...)

**High-level code**

```java
if (i == j)
    f = g + h;

f = f - i;
```

**MIPS assembly**

```mips
# $s0 = f, $s1 = g
# $s2 = h
# $s3 = i, $s4 = j
bne $s3, $s4, L1
add $s0, $s1, $s2
L1: sub $s0, $s0, $s3
```
We use the *unconditional branch* (i.e., \( j \)) to skip the "*else*" subtask if the "*if*" subtask is the correct one.

### High-level code

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

### 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:
```
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**

```plaintext
// 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**

```plaintext
# $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:
```
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

```
# $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:
```
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**

\[ $t1 = \text{s0} < \text{t0} \, ? \, 1:0 \]
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
```
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

```
# 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 = 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
algorithm 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>7FFFFFFF8</td>
<td></td>
<td>7FFFFFFF8</td>
<td></td>
</tr>
<tr>
<td>7FFFFFFF4</td>
<td></td>
<td>7FFFFFFF4</td>
<td></td>
</tr>
<tr>
<td>7FFFFFFF0</td>
<td></td>
<td>7FFFFFFF0</td>
<td></td>
</tr>
</tbody>
</table>

Two words pushed on the stack

$sp$
The Stack: Code Example

MIPS assembly

diffofsums:
  addi $sp, $sp, -12  # allocate space on stack to store 3 registers
  sw  $s0, 8($sp)    # save $s0 on stack
  sw  $t0, 4($sp)    # save $t0 on stack
  sw  $t1, 0($sp)    # save $t1 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  $t1, 0($sp)    # restore $t1 from stack
  lw  $t0, 4($sp)    # restore $t0 from stack
  lw  $s0, 8($sp)    # restore $s0 from stack
  addi $sp, $sp, 12  # deallocate stack space
  jr  $ra            # return to caller

- 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
The Stack: Nonpreserved Registers

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

- Instruction Set Architectures: LC-3 and MIPS
  - Operate instructions
  - Data movement instructions
  - Control instructions

- Instruction formats

- Addressing modes

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