Instruction set architecture

- What is an instruction set architecture (ISA)?
  - It is all of the programmer-visible components and operations of the computer
  - The ISA provides all the information needed for someone to write a program in machine language
  - Or translate from a high-level language to machine language
Instruction set architecture

- Memory organization
  - Address space (how many locations can be addressed?)
  - Addressability (how many bits per location?)
- Register set
  - How many instructions?
  - What size?
  - How are they used?
- Instruction set
  - Opcodes
  - Data types
  - Addressing modes

LC-3 memory

- Address space: $2^{16}$ locations
  - Address bus: 16 bits
- Addressability: 16 bits per location
  - Data bus: 16 bits
- Access time: several clock cycles
- Volatile
  - Loses content at power off
Registers: GPRs \( 8 \times 16 = 2^8 \times 2^4 = 2^{12} = 128 \)

- 8 general-purpose registers (GPUs) in the CPU’s register file
- Address space: \(2^3=8\) locations
- Addressability: 16 bits per register
- Access time: 1 clock cycle
- Volatile
  - Lose content at power off

LC-3 special registers

- PC: Program Counter
  - Points to memory address of next instruction to execute
- IR: Instruction Register
  - Stores current instruction
- MAR: Memory Address Register
  - Address of current memory access
- MDR: Memory Data Register
  - Data to write to or read from memory
- Condition codes (CC) register
  - N, Z, P
  - All are 16 bits except CC (3 bits)
Instructions

- What do instructions look like?

Instruction set architecture

- Opcodes
- Data types
- Addressing modes
Instruction set architecture

- Opcodes
  - 16 opcodes (1 unused/reserved)
  - Operate (Logical or Arithmetic) instructions:
    - ADD, AND, NOT
  - Data movement instructions:
    - LD, LDI, LDR
    - ST, STR, STI
    - LEA
  - Control instructions:
    - BR, JSR/JSRR, JMP, RTI, TRAP
  - Some opcodes set/clear condition codes, based on result:
    - N = negative (< 0)
    - Z = zero
    - P = positive (> 0)

---

Instruction set architecture

- Data Types
  - 16-bit 2’s complement integer
- Addressing Modes
  - How operands are specified
  - Or how the next instructions to execute is specified
  - Architecture-specific
  - An instruction can use several addressing modes
Are these enough?

- Are ADD, AND, and NOT enough?
- With only ADD, AND, and NOT:
  - How do we subtract?
    \[ A - B = A \text{ ADD} (\text{NOT} \ B \text{ ADD} 1) \]
  - How do we OR?
    \[
    \begin{align*}
    A + B &= \overline{\overline{A} \cdot \overline{B}} \\
    \overline{A} + \overline{B} &= \overline{A} \cdot \overline{B}
    \end{align*}
    \]
    \[ A \text{ OR} B = \text{NOT} \left( (\text{NOT} \ A) \text{ AND} (\text{NOT} \ B) \right) \]

Are these enough?

- With only ADD, AND, and NOT: \[ R_5 \leftarrow R_2 \]
  - How do we copy from one register to another?
    \[ \text{ADD} \ R_5, \ R_5, \ \text{-} R_5 \]
    \[ \text{ADD} \ R_5, \ R_5, \ R_2 \]
  - How do we initialize a register to zero?
    \[ \text{AND} \ R_x, \ R_x, \ 0 \]
Addressing modes

An exhaustive list of the LC-3 addressing modes

- Immediate
- Register
- PC-Relative
- Base+Offset
- Memory-indirect

Addressing modes: Immediate

- Immediate: a numeric value embedded in the instruction is the actual operand.
  - Data movement instructions: ADD, AND, LEA
  - Control flow instructions: none
  - You can tell an instruction uses this addressing mode when…
    * A constant value is explicitly specified in one of the named instructions

ADD R5, R5, #1
**Instruction: ADD / AND**

Note: Immediate field is sign-extended.

Addressing mode(s): REGISTER and IMMEDIATE

**Addressing modes: Register**

- Register: a source or destination operand is specified as content of one of the registers R0 – R7.
  - Data movement instructions: ADD, AND, NOT, LD, LDI, LDR, LEA, ST, STI, STR
  - Control flow instructions: JMP, RET, JSRR
  - You can tell an instruction uses this addressing mode when…
    - A register is explicitly specified in the instruction
**Instruction: NOT**

- **NOT RS, R2**
- Notes:
  - Works only with registers
  - Src and Dst can be the same register

**Addressing mode(s): REGISTER**

---

**Instruction: ADD / AND**

- **ADD RS, RS, R2**
- **AND RS, RS, R2**

**ADD**

```
0 0 0 1  Dst  Src1 0 0 0  Src2
```

**AND**

```
0 1 0 1  Dst  Src1 0 0 0  Src2
```

**Register File**

**Addressing mode(s): REGISTER**

This zero means "register mode"
Addressing modes: PC-Relative

- The problem: We want to specify address directly in the instruction
- But an address is 16 bits, and so is an instruction
- After subtracting 4 bits for opcode and 3 bits for register, we have only 9 bits available for address

9 bits allows the offset range to be
- \(-256 \leq \text{offset} \leq +255\)

We can now form any address \(X\), such that
- \((\text{PC} - 256) \leq X \leq (\text{PC} + 255)\)

Remember that the PC is incremented before the instruction is executed
Addressing modes: PC-Relative

- PC-relative: a data or instruction memory location is specified as an offset relative to the incremented PC
  - Data movement instructions: **LD, ST**
  - Control flow instructions: **BR, JSR**
  - You can tell an instruction uses this addressing mode when...
    - The instruction is one of the ones named above

Instruction: **LD** (Load Data)  \( \text{LD } R_S, <\text{offset}> \)

Addressing mode(s):
PC-RELATIVE (and REGISTER)
\[ \text{DR} = M[\text{PC+}\text{SX}(\text{PCoffset9})] \]
Instruction: ST (Store Data)

ST Rs, <Offset>

Addressing mode(s):
PC-RELATIVE (and REGISTER)
M[PC+X(PC+offset9)] = SR

Instruction: LD / ST

<table>
<thead>
<tr>
<th>x3000</th>
<th>.</th>
<th>.</th>
<th>.</th>
<th>x3010</th>
<th>LD</th>
<th>R0, BOB</th>
<th>R0 = xFFFF (-1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3011</td>
<td>LD</td>
<td>R1, BILL</td>
<td>R1 = x0002 (2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x3012</td>
<td>ADD</td>
<td>R2, R0, R1</td>
<td>R2 = x0001</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x3013</td>
<td>ST</td>
<td>R2, RESULT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| x3020 | BOB | .FILL | xFFFF | | |
| x3021 | BILL | .FILL | 0x0002 | | |
| x3022 | RESULT | .FILL | 0x0000 | | |

directives
Addressing Modes: Base+Offset

- Problem: With PC-relative mode, we can only address data within 256 words of the instruction
  - What about the rest of memory? How do we access it?
- Solution: Use a register to generate a full 16-bit address
  - 4 bits for opcode
  - 3 bits for source / destination register
  - 3 bits for base register
  - Remaining 6 bits are used as a signed offset
  - Offset is sign-extended before adding to base register

Addressing Modes: Base+Offset

- Base+offset: a data or instruction memory location is specified as a signed offset from a base register.
  - Data movement instructions: LDR, STR
  - Control flow instructions: TRAP, RTI
  - You can tell an instruction uses this addressing mode when…
    - The instruction is one of the ones named above
**Instruction: LDR (Load Register)**

LDR R5, R2, 0  
R5 ← [R2+0]

Addressing mode(s):
BASE+OFFSET (and REG.)  
DR = M[Base+SX(IR[5:0])]

**Instruction: STR (Store Register)**

STR R5, R2, 0  
R5 ← M[R2+0]

Addressing mode(s):
BASE+OFFSET (and REG.)  
M[Base+SX(IR[5:0])] = Src
Addressing modes: Memory-Indirect

- Double indirection
  - Use indirect addressing mode to point to a pointer
- First, address is generated from PC and IR
  - Just like PC-relative addressing
- Next, read from that address
  - But the memory location holds data that is actually another address (it’s a pointer)
- Use the retrieved address to load or store
  - Depending on the instruction

Addressing modes: Memory-Indirect

- Memory-indirect: a data memory location (specified as PC-RELATIVE) is a pointer to a data memory location.
  - Data flow instructions: LDI, STI
  - Control flow instructions: none
  - You can tell an instruction uses this addressing mode when…
    * The instruction says “indirect” in the title
Instruction: **LDI** (Load Indirect)

Addressing mode(s):
MEMORY INDIRECT (and REG)

$$DR = M[PC+SX(PC_{offset9})]$$

Instruction: **STI** (Store Indirect)

Addressing mode(s):
MEMORY INDIRECT (and REG)

$$DB = M[PC+SX(PC_{offset9})]$$
**Instruction: LDI/STI**

<table>
<thead>
<tr>
<th>x3000</th>
<th>.</th>
<th>.</th>
<th>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3010</td>
<td>LDI R0, BOB_P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x3011</td>
<td>LDI R1, BILL_P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x3012</td>
<td>ADD R2, R0, R1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x3013</td>
<td>. .</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
x301D RES_P .FILL 0x3022
x301E BOB_P .FILL 0x3020
x301F BILL_P .FILL BILL
x3020 BOB .FILL 0xFFFF
x3021 BILL .FILL 0x0002
x3022 RESULT .FILL 0x0000
```

**Instruction: LEA (Load Effective Address)**

- A strange instruction...
- Computes address of a label as PC-relative
  - PC plus signed offset
- Stores the result in a register
- Note: The address is stored in the register
  - Not the contents of the memory location
- Addressing modes used
  - Immediate
  - Register
Instruction: **LEA (Load Effective Address)**

Addressing mode(s):
- IMMEDIATE (and REG.)

$$ \text{Dest} = \text{SX(PC offset)} $$

---

**x3000**
- LEA R2, BOB
- LDR R0, R2, 0

**x3010**
- BOB .FILL 0xFFFF

**x3011**
- BILL .FILL 0x0002

**x3020**
- R2 = x3020
- R0 = xFFFF

**x3021**
- BOB .FILL 0xFFFF
- BILL .FILL 0x0002
### Addressing modes summary

- **IMMEDIATE**: a numeric value embedded in the instruction is the actual operand. Data: `ADD, AND, LEA`.
- **REGISTER**: a source or destination operand is specified as content of one of the registers R0–R7. Data: `ADD, AND, NOT, LD, LDI, LDR, LEA, ST, STI, STR`. Control: `JMP, RET, JSRR`.
- **PC-RELATIVE**: a data or instruction memory location is specified as an offset relative to the incremented PC. Data: `LD, ST`. Control: `BR, JSR`.
- **BASE+OFFSET**: a data or instruction memory location is specified as a signed offset from a base register. Data: `LDR, STR`. Control: `TRAP, RTI`.
- **MEMORY-INDIRECT**: a data memory location (specified as PC-RELATIVE) is a pointer to a data memory location. Data: `LDI, STI`.

### Which addressing mode is best?

- Which addressing mode gives you the greatest range?
  - Access most memory
- Need all the bits…
- Base+offset
- Memory indirect
### Operate instructions

- Only three operations
  - ADD, AND, NOT
- Source and destination operands are registers
  - These instructions do not reference memory
- Addressing modes
  - NOT uses only register addressing
  - ADD and AND can use either...
    - Register addressing (when all operands are registers)
    - Immediate and register addressing (where one of the source operands is an explicitly number encoded within the instruction)

---

### Full LC-3 Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>000</td>
<td>Source RA, Destination RB</td>
</tr>
<tr>
<td>AND</td>
<td>001</td>
<td>Source RA, Destination RB</td>
</tr>
<tr>
<td>NOT</td>
<td>010</td>
<td>Source RA</td>
</tr>
<tr>
<td>BR</td>
<td>100</td>
<td>Program Counter (PC)</td>
</tr>
<tr>
<td>JMP</td>
<td>101</td>
<td>Source RA, Program Counter (PC)</td>
</tr>
<tr>
<td>JER</td>
<td>110</td>
<td>Source RA, Program Counter (PC)</td>
</tr>
<tr>
<td>JERB</td>
<td>111</td>
<td>Source RA, Program Counter (PC)</td>
</tr>
<tr>
<td>LD</td>
<td>0000</td>
<td>Source RA, Program Counter (PC)</td>
</tr>
<tr>
<td>LDS</td>
<td>0001</td>
<td>Source RA, Program Counter (PC)</td>
</tr>
<tr>
<td>LDR</td>
<td>0010</td>
<td>Source RA, Program Counter (PC)</td>
</tr>
<tr>
<td>LDRR</td>
<td>0011</td>
<td>Source RA, Program Counter (PC)</td>
</tr>
<tr>
<td>LDRS</td>
<td>0100</td>
<td>Source RA, Program Counter (PC)</td>
</tr>
<tr>
<td>LDRS</td>
<td>0101</td>
<td>Source RA, Program Counter (PC)</td>
</tr>
<tr>
<td>LDRS</td>
<td>0110</td>
<td>Source RA, Program Counter (PC)</td>
</tr>
<tr>
<td>LDRS</td>
<td>0111</td>
<td>Source RA, Program Counter (PC)</td>
</tr>
<tr>
<td>LDRS</td>
<td>1000</td>
<td>Source RA, Program Counter (PC)</td>
</tr>
<tr>
<td>LDRS</td>
<td>1001</td>
<td>Source RA, Program Counter (PC)</td>
</tr>
<tr>
<td>LDRS</td>
<td>1010</td>
<td>Source RA, Program Counter (PC)</td>
</tr>
<tr>
<td>LDRS</td>
<td>1011</td>
<td>Source RA, Program Counter (PC)</td>
</tr>
<tr>
<td>LDRS</td>
<td>1100</td>
<td>Source RA, Program Counter (PC)</td>
</tr>
<tr>
<td>LDRS</td>
<td>1101</td>
<td>Source RA, Program Counter (PC)</td>
</tr>
<tr>
<td>LDRS</td>
<td>1110</td>
<td>Source RA, Program Counter (PC)</td>
</tr>
<tr>
<td>LDRS</td>
<td>1111</td>
<td>Source RA, Program Counter (PC)</td>
</tr>
</tbody>
</table>

---

The Full LC-3 Instruction Set includes more than just the Operate instructions. It also includes Branch, Jump, Load, Store, and Return instructions, each with specific formats and descriptions. The table above provides a snapshot of the Operate instructions, which are characterized by their simplicity in terms of addressing modes and the absence of memory references.
Data movement instructions

- Load - read data from memory to register
  - LD: PC-relative mode
  - LDR: base+offset mode
  - LDI: memory-indirect mode
- Store - write data from register to memory
  - ST: PC-relative mode
  - STR: base+offset mode
  - STI: memory-indirect mode
- Load effective address - compute address, save in register
  - LEA: immediate mode
  - Note: does not access memory

Control instructions

- Control instructions change the PC
- Used to alter the sequence of instructions
- Conditional Branch
  - If specified condition is true, branch is taken
    - Signed offset is added to the PC
  - Otherwise, branch is not taken
    - PC is not changed; falls through to next instruction in sequence
Control instructions

- Unconditional Branch (called a jump)
  - Always changes the PC

- Trap
  - It’s a type of exception
  - Changes the PC to the address of an operating system service routine
    - E.g., getc
  - Returns control to the next instruction after the trap when it is finished

Condition codes

- LC-3 has three condition code bits:
  - N – negative
  - Z – zero
  - P -- positive (greater than zero)

- Set by any instruction that writes a value to a register (ADD, AND, NOT, LD, LDR, LDI, LEA)
  - They have + next to them in the instruction list
  - Exactly one is set at all times
  - Set based on the last instruction that altered a register
Branch instruction

- Branch specifies one or more condition codes.
- If the set bit is specified, the branch is taken.
  - PC-relative addressing
  - Target address is formed by adding signed offset to the incremented PC
- Syntax: \( \text{BR}[n|z|p] \)
  - \( \text{BRnzp} = \text{unconditional branch (always)} \)
- If the branch is not taken, the next sequential instruction is executed
- Note: Target must be within 256 words of \( \text{BR} \) instruction (Why?)

Instruction: \( \text{BR} \) (Branch (if…))

Addressing mode(s):
- \( \text{PC-RELATIVE} \)
- \( \text{PC} = \text{PC} + \text{SX(POffset9)} \)
Example: Using a branch

- Compute the sum of 12 integers

```plaintext
Compute the sum of 12 integers

sum = 0
for (i=0; i<12; i++) {
    sum = sum + R1[i];
}
sum: .fill 0
```

---

Example: Using a branch

```
R1 ← x3100
R3 ← 0
R2 ← 12

R2=0?
```

- YES
  - R4 ← M[R1]
  - R3 ← R3+R4
  - R1 ← R1+1
  - R2 ← R2-1

- NO
  - ADD R2, R1, -12
  - BRZP AWAY
  - LD R3, SUM
  - ADD R3, R5, R1
  - ST R3, SUM
  - ADD R1, R1, 1
  - BRZP LOOP
Example: Binary to assembly

<table>
<thead>
<tr>
<th>Addr</th>
<th>Instruction Bits</th>
<th>Instruction</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3000</td>
<td>1 1 1 0 0 0 1 0 0 0 0 1 0 1 0</td>
<td>LEA R1, R1, R1, R1</td>
<td>R, I</td>
</tr>
<tr>
<td>x3001</td>
<td>0 1 0 1 0 1 0 1 0 1 1 0 0 0 0 0</td>
<td>AND R3, R3, #0</td>
<td>R, I</td>
</tr>
<tr>
<td>x3002</td>
<td>0 1 0 1 0 1 0 1 0 1 0 0 0 0 0</td>
<td>AND R2, R2, #0</td>
<td></td>
</tr>
<tr>
<td>x3003</td>
<td>0 0 0 1 0 1 0 0 1 0 1 0 1 1 0 0</td>
<td>ADD R2, R2, #12</td>
<td></td>
</tr>
<tr>
<td>x3004</td>
<td>0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 1</td>
<td>BRz R1, R1, #0</td>
<td></td>
</tr>
<tr>
<td>x3005</td>
<td>0 1 0 1 0 0 0 1 1 0 0 0 0 0 0</td>
<td>LDR R4, R1, #0</td>
<td>R, R</td>
</tr>
<tr>
<td>x3006</td>
<td>0 0 1 0 1 1 1 0 0 1 0 0 0 0 0 0</td>
<td>ADD R3, R4, #0</td>
<td></td>
</tr>
<tr>
<td>x3007</td>
<td>0 0 1 0 1 1 1 0 0 1 0 0 0 0 0 0</td>
<td>ADD R1, R1, #1</td>
<td>R, I</td>
</tr>
<tr>
<td>x3008</td>
<td>0 0 1 0 1 0 1 0 1 1 1 1 1 1</td>
<td>ADD R2, R2, #12</td>
<td></td>
</tr>
<tr>
<td>x3009</td>
<td>0 0 1 0 1 1 1 1 1 1 1 1 1 1 0 1</td>
<td>BRnzp L1</td>
<td></td>
</tr>
</tbody>
</table>

LC-3 source code

```
.orig x3000
LEA R1, ARRAY
AND R3, R3, #0
AND R2, R2, #0
ADD R2, R2, #12
LOOP
BRz DONE
LDR R4, R1, #0
ADD R3, R3, R4
ADD R1, R1, #1
ADD R2, R2, #1
BRNZP LOOP
DONE
HALT
ARRAY .blkw #12 #3
.end
```

Note: The source code and binary bits are aligned to show the correspondence between the two representations.
**Instruction: JMP (Jump)**

- Jump is an unconditional branch
  - Always taken
  - Allows any target address
  - Specified by a register

**Addressing mode(s):**
- REGISTER
  - PC = Base

Wouldn’t it be nice...

- Wouldn’t it be nice to have a way to reuse code?
- Jump to a location, execute the code, and return
- Be able to call it whenever we want
- From anywhere
- And have it remember where we came from
Instruction: JSR (Jump Subroutine)

- Jump to subroutine
  ◆ Used to implement function calls
- Save current (incremented) PC in R7
  ◆ Used to return from subroutine
- Increment PC by the 11-bit signed offset

Instruction: JSRR (Jump Subroutine Register)

- Jump to subroutine specified by address in register
  ◆ Used to implement function calls
- Save the current (incremented) PC in R7
  ◆ Used to return from subroutine
- PC gets the value specified by a register called BaseR

Addressing mode(s):
- PC-RELATIVE
  \[ PC = PC + SX(PC_{offset11}) \]
- REGISTER
  \[ PC = \text{BaseR} \]

Addressing mode(s):
- REGISTER
  \[ PC = \text{BaseR} \]
  \[ R7 = \text{old PC} \]
Instruction: **RET** (Return)

- Return from subroutine
  - Replaces the PC with the content of the return address register R7
  - **RET** is just a different mnemonic for…
    - *JMP R7*
- Addressing mode is a **special case** of register addressing with R7

![Addressing mode(s): REGISTER (special case with REGISTER = R7)]

```
RET 1100 0001 11000000
```

Instruction: **RTI** (Return from Interrupt)

- Return from interrupt
  - Replaces the PC with the content the memory location pointed to by R6 (the stack pointer)
- Wait, what?
  - Return address is on the stack!

![Addressing mode(s): BASE+OFFSET (Special case with BASE = R6 and OFFSET = 0)]

```
RTI 1100 0001 11000000
```
Instruction: TRAP

- Calls a service routine, identified by 8-bit trap vector
- When routine is complete, PC is re-set to the instruction following the TRAP call
- Base+offset

Addressing mode(s):
- BASE+OFFSET
  \( PC = M[R6] \)

<table>
<thead>
<tr>
<th>Vector</th>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>x23</td>
<td>input a character from the keyboard TRAP x23 = GETC</td>
</tr>
<tr>
<td>x21</td>
<td>output a character to the monitor TRAP x21 = OUT</td>
</tr>
</tbody>
</table>

Instruction: TRAP

- Execute code at a TRAP subroutine
  - Used to implement operating-system-like function calls
  - Just like JSR/JSRR, TRAP saves the current incremented PC in R7
  - Used to return from the TRAP subroutine
- When called, the PC gets the value specified in a trap vector
  - Located in memory at addresses x20—x24 (the trap numbers)
- Addressing mode is a special case of base+offset
  - No base register and unsigned offset

Addressing mode(s):
- BASE+OFFSET
  \( PC = M[ZX(trapvect8)] \)
Example: Char count

- **Goal**
  - Count the occurrences of a character in an array

- **Specifications**
  - Input: Read character from the keyboard
  - Output: Display the number of occurrences of a character found in the array

- **Assumptions**
  - Fewer than 10 occurrences

- **Implementation details**
  - Program begins at location x3000
  - Load each character from an array
    - An array is a sequence of memory locations
    - Starting address is immediately after the last line of the program
  - If array character equals input character, increment counter
  - End of array is indicated by a special ASCII value: EOT (x04)

---

Char count: Flow chart

- **Count = 0**
  - *(R2 = 0)*

- **Ptr = 1st character of array**
  - *(R3 = M[x3012])*  

- **Input char from keybd**
  - *(TRAP x23)*

- **Load char from array**
  - *(R1 = M[R3])* 

- **Done?**
  - *(R1 ?= EOT)*

- **Incr Count**
  - *(R2 = R2 + 1)*

- **Match?**
  - *(R1 ?= R0)*

- **Print count**
  - *(TRAP x21)*

- **HALT**
  - *(TRAP x25)*
### Char count (1/2)

<table>
<thead>
<tr>
<th>Addr</th>
<th>Instruction Bits</th>
<th>Instruction</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3000</td>
<td>0101010010100000</td>
<td>AND</td>
<td></td>
</tr>
<tr>
<td>x3001</td>
<td>0010011000010000</td>
<td>LD</td>
<td></td>
</tr>
<tr>
<td>x3002</td>
<td>1111000000100011</td>
<td>TRAP</td>
<td></td>
</tr>
<tr>
<td>x3003</td>
<td>0110001011000000</td>
<td>LDR</td>
<td></td>
</tr>
<tr>
<td>x3004</td>
<td>0001100001111100</td>
<td>ADD</td>
<td></td>
</tr>
<tr>
<td>x3005</td>
<td>0000010000001000</td>
<td>BRz</td>
<td></td>
</tr>
<tr>
<td>x3006</td>
<td>1001001001111111</td>
<td>NOT</td>
<td></td>
</tr>
<tr>
<td>x3007</td>
<td>0001010001100001</td>
<td>ADD</td>
<td></td>
</tr>
<tr>
<td>x3008</td>
<td>0001001001000000</td>
<td>ADD</td>
<td></td>
</tr>
<tr>
<td>x3009</td>
<td>0000010100000000</td>
<td>BRnp</td>
<td></td>
</tr>
</tbody>
</table>

### Char count (2/2)

<table>
<thead>
<tr>
<th>Addr</th>
<th>Instruction Bits</th>
<th>Instruction</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>x300A</td>
<td>0001010010100001</td>
<td>ADD</td>
<td></td>
</tr>
<tr>
<td>x300B</td>
<td>0001011011000001</td>
<td>ADD</td>
<td></td>
</tr>
<tr>
<td>x300C</td>
<td>0110001011000000</td>
<td>LDR</td>
<td></td>
</tr>
<tr>
<td>x300D</td>
<td>0000111111101110</td>
<td>BRnzp</td>
<td></td>
</tr>
<tr>
<td>x300E</td>
<td>0010000000000100</td>
<td>LD</td>
<td></td>
</tr>
<tr>
<td>x300F</td>
<td>0001000000000010</td>
<td>ADD</td>
<td></td>
</tr>
<tr>
<td>x3010</td>
<td>1111000000100001</td>
<td>TRAP</td>
<td></td>
</tr>
<tr>
<td>x3011</td>
<td>1111000000100101</td>
<td>TRAP</td>
<td></td>
</tr>
<tr>
<td>X3012</td>
<td>0011000000010100</td>
<td>- Starting Address of Array</td>
<td></td>
</tr>
<tr>
<td>x3013</td>
<td>0000000000110000</td>
<td>Data</td>
<td></td>
</tr>
</tbody>
</table>
Recommended exercises

- Ex 5.4, 5.6, 5.8, 5.9, 5.11, 5.12
- Ex 5.16, 5.30, 5.32, 5.33, 5.40, 5.41

Especially interesting:
- Ex 5.13, 5.14, 5.15, 5.22, 5.23, 5.25
- Ex 5.26 and 5.42 (on ISA design)
- Ex 5.31

Sample Code

```asm
.orig  x3000
AND R0, R0, #0
LD R1, UNO
LD R2, DUE
INTEL ADD R0, R0, R2
ADD R1, R1, #-1
BRZ AMD
BRNZP INTEL
AMD ST R0, TRE
HALT
UNO .FILL X000E
DUE .FILL X0003
TRE .FILL XFFFF
.end
```

What does this program do?
Sample code in the simulator

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3000</td>
<td>0101000000100000</td>
<td>x3020 AND R0, R0, #0</td>
</tr>
<tr>
<td>x3001</td>
<td>0010001000000111</td>
<td>x2207 LD R1, UND</td>
</tr>
<tr>
<td>x3002</td>
<td>0010001000000011</td>
<td>x2407 LD R3, DUE</td>
</tr>
<tr>
<td>x3003</td>
<td>0001000000000016</td>
<td>x1902 INTEL AND R0, R0, R2</td>
</tr>
<tr>
<td>x3004</td>
<td>0001001001111111</td>
<td>x127F ADD R1, R1, #1</td>
</tr>
<tr>
<td>x3005</td>
<td>0000010000000001</td>
<td>x6401 BRZ AND</td>
</tr>
<tr>
<td>x3006</td>
<td>0000011111111110</td>
<td>x1FFC SUBZP INTEL</td>
</tr>
<tr>
<td>x3007</td>
<td>0110100000000011</td>
<td>x0000 AND R1, R0, DUE</td>
</tr>
<tr>
<td>x3008</td>
<td>1111000000101010</td>
<td>x7025 TRAP HALT</td>
</tr>
<tr>
<td>x3009</td>
<td>0000000000011111</td>
<td>x0002E UNO NOP</td>
</tr>
<tr>
<td>x300A</td>
<td>0000000000000011</td>
<td>x0003 DUE NOP</td>
</tr>
<tr>
<td>x300B</td>
<td>0000000001010101</td>
<td>x002A THE NOP</td>
</tr>
<tr>
<td>x300C</td>
<td>0000000000000000</td>
<td>x0000 NOP</td>
</tr>
</tbody>
</table>

SimpleMult.obj | 89 instructions executed | ide