LC-3
Instruction Set Architecture
(Ch5)
AND bitwise

0110
0000
0101
0000
\texttt{R1} = -1

\texttt{HALT}

\texttt{printf}\n
\texttt{Num \div 10 = digit}
Instruction Set Architecture

ISA is all of the *programmer-visible* components and operations of the computer.

- memory organization
  - address space -- how may locations can be addressed?
  - addressability -- how many bits per location?
- register set
  - how many? what size? how are they used?
- instruction set
  - opcodes
  - data types
  - addressing modes

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).
Memory vs. Registers

Memory
- address space: $2^{16}$ locations (16-bit addresses)
- addressability: 16 bits

Registers
- temporary storage, accessed in a single machine cycle
  - accessing memory generally takes longer than a single cycle
- eight general-purpose registers: R0 - R7
  - each is 16 bits wide
  - how many bits to uniquely identify a register?
- other registers
  - not directly addressable, but used/effect by instructions
  - PC (program counter), condition codes
Instruction Set

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

Data Types
- 16-bit 2’s complement integer

Addressing Modes
- How is the location of an operand specified?
- non-memory addresses: **immediate, register**
- memory addresses: **PC-relative, indirect, base+offset**
Operate Instructions

Only three operations: **ADD, AND, NOT**

Source and destination operands are registers
- These instructions *do not* reference memory.
- **ADD** and **AND** can use “immediate” mode, where one operand is hard-wired into the instruction.

Will show dataflow diagram with each instruction.
- illustrates *when* and *where* data moves to accomplish the desired operation
Note: \( \text{Src} \) and \( \text{Dst} \) could be the same register.

Note: works only with registers.
ADD/AND

ADD
0 0 0 1 | Dst | Src1 | 0 0 0 | Src2

AND
0 1 0 1 | Dst | Src1 | 0 0 0 | Src2

This zero means "register mode"
ADD/AND

ADD

| 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  |  | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|
| 0  | 0  | 0  | 1  |   | Dst| Src1| 1  |   |   |   |   |   |   |   |   |   |   |   |   |

AND

| 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  |  | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|
| 0  | 1  | 0  | 1  |   | Dst| Src1| 1  |   |   |   |   |   |   |   |   |   |   |   |   |

ADD R1, R2, 3

Note: Immediate field is sign-extended.

This one means "immediate mode"
Data Movement Instructions

Load -- read data from memory to register
- **LD**: PC-relative mode
- **LDR**: base+offset mode
- **LDI**: indirect mode

Store -- write data from register to memory
- **ST**: PC-relative mode
- **STR**: base+offset mode
- **STI**: indirect mode

Load effective address -- compute address, save in register
- **LEA**: immediate mode
- *does not access memory*
Addressing Modes

- How memory is addressed.
- Different instructions use different addressing modes.
- Some instructions support more than one addressing mode.
LC-3 Addressing Modes

• PC-Relative
  – Address is a displacement from PC

• Indirect
  – Use PC-Relative to get address from memory

• Base plus Offset
  – Use contents of a register as base address and add offset to find address (most common for load/store architectures)
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.
The Solution:
Use the 9 bits as a *signed offset* from the current PC.

9 bits allows the offset range to be:

\[-256 \leq \text{offset} \leq +255\]

We can now form any address \( X \), such that:

\[(PC - 256) \leq X \leq (PC + 255)\]

Remember that the PC is incremented as part of the **FETCH** phase; This is done before the **EVALUATE ADDRESS** stage.
LD (Load Data)

LD

PC

Sext

IR[8:0]

Instruction Reg

Register File

Dst

Memory

PCoffset9

MAR

MDR

CMPE-012/L
Linker Table

ST (Store Data)

<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>ST</td>
<td>0 0 1 1</td>
<td>Src</td>
<td>PCoffset9</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>

PC

Instruction Reg

Sext

IR[8:0]

Register File

Src

Memory

MAR

MDR

LDR R1, F00
Indirect

The 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 #1:

- Read address from memory location, then load/store to that address.

First address is generated from PC and IR (just like PC-relative addressing), then content of that address is used as target for load/store.
L1) I RO, bar
6 ar = pc-relative
MDR = FOOD
MAR = FOOD
RO = M[FOOD]
Base + Offset

Remember The Problem:
With PC-relative mode, can only address data within 256 words of the instruction.
  – What about the rest of memory? How do we access it?
Solution #2:

- Use a register to generate a full 16-bit address.

4 bits for opcode, 3 bits for src/dest register, 3 bits for base register – the remaining 6 bits are used as a signed offset.

- Offset is sign-extended before adding to base register.
Load Effective Address

Computes address like PC-relative (PC plus signed offset) and stores the result into a register.

Note: The address is stored in the register, not the contents of the memory location.
LEA (Immediate)

LEA 1110 Dst PC offset 9

PC

Sext

Instruction Reg

Register File

Dst

CMPE-012/L
Control Instructions

Used to alter the sequence of instructions. This is done by changing the PC.

Conditional Branch

- branch is *taken* if a specified condition is true
  - signed offset is added to PC to yield new PC
- else, the branch is *not taken*
  - PC is not changed, points to the next sequential instruction
Unconditional Branch (or Jump)

- always changes the PC

TRAP

- changes PC to the address of an OS “service routine”
- routine will return control to the next instruction (after TRAP) when 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)

Exactly one will be set at all times
– Based on the last instruction that altered a register
Branch Instruction

BRNP

• Branch specifies one or more condition codes.
• If the set bit is specified, the branch is taken.
  – PC-relative addressing is used
  – target address is made by adding signed offset (IR[8:0]) to current PC.
If the branch is not taken, the next sequential instruction is executed.

- Note: PC has already been incremented by FETCH stage.
- Note: Target must be within 256 words of BR instruction.
BR (PC-Relative)

BR 0 0 0 0 n z p PCoffset9

BRz

PC

Logic

N Z P

IR[11:9]

Sext

IR[8:0]

Instruction Reg

PCMUX

taken

Interupts
```
N 2 p

msb ≤ 4

\[ \text{everything is zero} \]

1

Bit 11 AND N2

Bit 10 AND \( \lfloor \frac{2}{p} \rfloor \)
```

JMP

Jump is an unconditional branch -- *always* taken.
- Target address is the contents of a register.
- Allows any target address.
TRAP

1 1 1 1 0 0 0 0 trapvect8

Calls a service routine, identified by 8-bit “trap vector.”

<table>
<thead>
<tr>
<th>Vector</th>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>x23</td>
<td>input a character from the keyboard</td>
</tr>
<tr>
<td>x21</td>
<td>output a character to the monitor</td>
</tr>
<tr>
<td>x25</td>
<td>halt the program</td>
</tr>
</tbody>
</table>

When routine is done, PC is set to the instruction following TRAP.
Another Example

Count the occurrences of a character in an array

- Program begins at location x3000
- Read character from keyboard
- Load each character from an array
  - An array is a sequence of memory locations
  - Starting address of array is stored in the memory location immediately after the program
- If array character equals input character, increment counter
- End of array is indicated by a special ASCII value: EOT (x04)
- At the end, print the number of characters and halt
  (lets assume there will be less than 10 occurrences of the character)
## Program (page 1 of 2)

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Bits</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3000</td>
<td>0 1 0 1 0 1 0 0 1 0 1 0 1 0 0 0 0 0 0</td>
<td>AND</td>
<td>R2 ← 0 (counter)</td>
</tr>
<tr>
<td>x3001</td>
<td>0 0 1 0 0 1 1 0 0 0 0 1 0 0 0 0 0</td>
<td>LD</td>
<td>R3 ← M[x3012] (ptr)</td>
</tr>
<tr>
<td>x3002</td>
<td>1 1 1 1 0 0 0 0 0 0 0 1 0 0 0 1 1</td>
<td>TRAP</td>
<td>Input to R0 (TRAP x23)</td>
</tr>
<tr>
<td>x3003</td>
<td>0 1 1 0 0 0 1 0 1 1 0 0 0 0 0 0 0</td>
<td>LDR</td>
<td>R1 ← M[R3]</td>
</tr>
<tr>
<td>x3004</td>
<td>0 0 0 1 1 0 0 0 0 0 1 1 1 1 0 0</td>
<td>ADD</td>
<td>R4 ← R1 - 4 (EOT)</td>
</tr>
<tr>
<td>x3005</td>
<td>0 0 0 0 0 1 0</td>
<td>BRz</td>
<td>If Z, goto x300E</td>
</tr>
<tr>
<td>x3006</td>
<td>1 0 0 1 0 0 1 0 0 1 1 1 1 1 1 1</td>
<td>NOT</td>
<td>R1 ← NOT R1</td>
</tr>
<tr>
<td>x3007</td>
<td>0 0 0 1 0 0 1 0 0 1 1 0 0 0 0 1</td>
<td>ADD</td>
<td>R1 ← R1 + 1</td>
</tr>
<tr>
<td>x3008</td>
<td>0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0</td>
<td>ADD</td>
<td>R1 ← R1 + R0</td>
</tr>
<tr>
<td>x3009</td>
<td>0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 1</td>
<td>BRnp</td>
<td>If N or P, goto x300B</td>
</tr>
</tbody>
</table>

**opcode**
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Bits</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>×300A</td>
<td>0 0 0 1 0 1 0 0 1 0 1 0 0 0 0 1</td>
<td>ADD</td>
<td>R2 ← R2 + 1</td>
</tr>
<tr>
<td>×300B</td>
<td>0 0 0 1 0 1 1 0 1 1 1 0 0 0 0 1</td>
<td>ADD</td>
<td>R3 ← R3 + 1</td>
</tr>
<tr>
<td>×300C</td>
<td>0 1 1 0 0 0 1 0 1 1 0 0 0 0 0 0</td>
<td>LDR</td>
<td>R1 ← M[R3]</td>
</tr>
<tr>
<td>×300D</td>
<td>0 0 0 0 1 1 1 1 1 1 1 1 0 1 1 0</td>
<td>BRnzp</td>
<td>Goto ×3004</td>
</tr>
<tr>
<td>×300E</td>
<td>0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0</td>
<td>LD</td>
<td>RO ← M[x3013]</td>
</tr>
<tr>
<td>×300F</td>
<td>0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0</td>
<td>ADD</td>
<td>RO ← RO + R2</td>
</tr>
<tr>
<td>×3010</td>
<td>1 1 1 1 0 0 0 0 0 0 1 0 0 0 0 1</td>
<td>TRAP</td>
<td>Print RO (TRAP ×21)</td>
</tr>
<tr>
<td>×3011</td>
<td>1 1 1 1 0 0 0 0 0 0 0 1 0 0 1 0 1</td>
<td>TRAP</td>
<td>HALT (TRAP ×25)</td>
</tr>
<tr>
<td>×3012</td>
<td>Starting Address of File</td>
<td></td>
<td></td>
</tr>
<tr>
<td>×3013</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0</td>
<td>Data</td>
<td>ASCII ×30 ('0')</td>
</tr>
</tbody>
</table>

**opcode**
LC-3
Data Path

Filled arrow = info to be processed

Unfilled arrow = control signal.
# LC-3 Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>0001</td>
<td>ADD</td>
</tr>
<tr>
<td>ADD</td>
<td>0001</td>
<td>ADD</td>
</tr>
<tr>
<td>AND</td>
<td>0101</td>
<td>AND</td>
</tr>
<tr>
<td>AND</td>
<td>0101</td>
<td>AND</td>
</tr>
<tr>
<td>NOT</td>
<td>1001</td>
<td>NOT</td>
</tr>
<tr>
<td>BR</td>
<td>0000</td>
<td>BR</td>
</tr>
<tr>
<td>JMP</td>
<td>1100</td>
<td>JMP</td>
</tr>
<tr>
<td>JSR</td>
<td>0100</td>
<td>JSR</td>
</tr>
<tr>
<td>JSRR</td>
<td>0100</td>
<td>JSRR</td>
</tr>
<tr>
<td>RET</td>
<td>1100</td>
<td>RET</td>
</tr>
<tr>
<td>LD</td>
<td>0010</td>
<td>LD</td>
</tr>
<tr>
<td>LDI</td>
<td>1010</td>
<td>LDI</td>
</tr>
<tr>
<td>LDR</td>
<td>0110</td>
<td>LDR</td>
</tr>
<tr>
<td>LEA</td>
<td>1110</td>
<td>LEA</td>
</tr>
<tr>
<td>ST</td>
<td>0011</td>
<td>ST</td>
</tr>
<tr>
<td>STI</td>
<td>1011</td>
<td>STI</td>
</tr>
<tr>
<td>STR</td>
<td>0111</td>
<td>STR</td>
</tr>
<tr>
<td>TRAP</td>
<td>1111</td>
<td>TRAP</td>
</tr>
<tr>
<td>RTI</td>
<td>1000</td>
<td>RTI</td>
</tr>
<tr>
<td>reserved</td>
<td>1101</td>
<td>reserved</td>
</tr>
</tbody>
</table>
Data Path Components

Global bus
- special set of wires that carry a 16-bit signal to many components
- inputs to the bus are “tri-state devices,” that only place a signal on the bus when they are enabled
- only one (16-bit) signal should be enabled at any time
  - control unit decides which signal “drives” the bus
- any number of components can read the bus
  - register only captures bus data if it is write-enabled by the control unit

Memory
- Control and data registers for memory and I/O devices
- memory: MAR, MDR (also control signal for read/write)
ALU

- Accepts inputs from register file and from sign-extended bits from IR (immediate field).
- Output goes to bus.
  - used by condition code logic, register file, memory

Register File

- Two read addresses (SR1, SR2), one write address (DR)
- Input from bus
  - result of ALU operation or memory read
- Two 16-bit outputs
  - used by ALU, PC, memory address
  - data for store instructions passes through ALU
PC and PCMUX

- There are three inputs to PC, controlled by PCMUX
  1. PC+1 – FETCH stage
  2. Address adder – BR, JMP
  3. bus – TRAP (discussed later)

MAR and MARMUX

- There are two inputs to MAR, controlled by MARMUX
  1. Address adder – LD/ST, LDR/STR
  2. Zero-extended IR[7:0] -- TRAP (discussed later)
Condition Code Logic
- Looks at value on bus and generates N, Z, P signals
- Registers set only when control unit enables them (LD.CC)
  - only certain instructions set the codes
    (ADD, AND, NOT, LD, LDI, LDR, LEA)

Control Unit – Finite State Machine
- On each machine cycle, changes control signals for next phase
  of instruction processing
  - who drives the bus? (GatePC, GateALU, ...)
  - which registers are write enabled? (LD.IR, LD.REG, ...)
  - which operation should ALU perform? (ALUK)
  - ...
- Logic includes decoder for opcode, etc.
Summary of ISA

- Instruction Set Architecture
- 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).
ADD R1, R2, R3
ADD R1, R2, 6
## LC-3 Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>0001</td>
<td>DR, SR1, 0, 00, SR2</td>
</tr>
<tr>
<td>ADD</td>
<td>0001</td>
<td>DR, SR1, 1, imm5</td>
</tr>
<tr>
<td>AND</td>
<td>0101</td>
<td>DR, SR1, 0, 00, SR2</td>
</tr>
<tr>
<td>AND</td>
<td>0101</td>
<td>DR, SR1, 1, imm5</td>
</tr>
<tr>
<td>NOT</td>
<td>1001</td>
<td>DR, SR, 111111</td>
</tr>
<tr>
<td>BR</td>
<td>0000</td>
<td>n, z, p, PCoffset9</td>
</tr>
<tr>
<td>JMP</td>
<td>1100</td>
<td>0, 00, BaseR, 000000</td>
</tr>
<tr>
<td>JSR</td>
<td>0100</td>
<td>1, PCoffset11</td>
</tr>
<tr>
<td>JSRR</td>
<td>0100</td>
<td>0, 00, BaseR, 000000</td>
</tr>
<tr>
<td>RET</td>
<td>1100</td>
<td>0, 00, 111, 000000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>0010</td>
<td>DR, PCoffset9</td>
</tr>
<tr>
<td>LDI</td>
<td>1010</td>
<td>DR, PCoffset9</td>
</tr>
<tr>
<td>LDR</td>
<td>0110</td>
<td>DR, BaseR, offset6</td>
</tr>
<tr>
<td>LEA</td>
<td>1110</td>
<td>DR, PCoffset9</td>
</tr>
<tr>
<td>ST</td>
<td>0011</td>
<td>SR, PCoffset9</td>
</tr>
<tr>
<td>STI</td>
<td>1011</td>
<td>SR, PCoffset9</td>
</tr>
<tr>
<td>STR</td>
<td>0111</td>
<td>SR, BaseR, offset6</td>
</tr>
<tr>
<td>TRAP</td>
<td>1111</td>
<td>0000, trapvect8</td>
</tr>
<tr>
<td>RTI</td>
<td>1000</td>
<td>00000000000000</td>
</tr>
</tbody>
</table>

**Reserved**

![Diagram](image)
Input and Output

How things get into and out of the CPU
Computer System
I/O: Connecting to Outside World

• So far, we’ve learned how to:
  – compute with values in registers
  – load data from memory to registers
  – store data from registers to memory

• But where does data in memory come from?

• And how does data get out of the system so that humans can use it?
Types of I/O devices characterized by:

- **behavior:** input, output, storage
  - input: keyboard, motion detector, network interface
  - output: monitor, printer, network interface
  - storage: disk, CD-ROM

- **data rate:** how fast can data be transferred?
  - keyboard: 100 bytes/sec
  - Spinning disk: 30 MB/s or more
  - SSD: over 200 MB/s to Samsung 960 claims 3,500 MB/s
  - network: 1 Mb/s - 1 Gb/s
Hospital
1 year
1 month
3+ears

Brain

7 info
functions
angles

90 wpm

Enter
I/O Devices

Keyboard

- User presses ‘A’ key -> ‘a’
- ASCII code is 0x61
- Keyboard sends this on wires
- 1 for start, 8-bits of data, 0 for stop
- ‘a’ is: 1011000010
- Buffer at computer catches these bits
Displays

• Character display works with the reverse process (sort of)
• Most displays today are “bit mapped”

Printers

• Just like a display but now being “printed” to paper, not a screen.
• Again, most printers are now “bit mapped” verses character.
Hard Disk

- A spinning disk (4600, 5200, 7200, 10000+ RPM)
- 20 – 8000 GB and growing FAST
- Magnetic and read/write (like tape)
- Both sides
- Usually a stack of platters
- Disk access

<table>
<thead>
<tr>
<th>Queuing</th>
<th>Seek</th>
<th>Rotation</th>
<th>Transfer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depends</td>
<td>10ms</td>
<td>10ms</td>
<td>1ms</td>
</tr>
</tbody>
</table>

- Electronic speeds are in the nanoseconds (10-9 sec)
- Disk speeds are in the milliseconds (10-3 sec)
- Why use a disk? cheap
- Replaced by SSD’s for most non-bulk storage
Questions

• How does CPU ask for a char to be printed?
• Which printer?
• Which display? Who’s?
• When is printer ready for the next char?
• When does keyboard have the next char?
• What about the million times slower?
LC-3 I/O

Uses the “TRAP” instruction to invoke the Operating System, which in turns talks to the hardware.

User specifies the type of operation desired by giving a code to the OS.

Ex. “TRAP x20” gets a character from the keyboard.
• Don’t use “JSR” because
  • OS doesn’t trust user to provide the correct address
  • Want to switch into OS mode, where more things are allowed
• CPU sees the “TRAP” instruction and uses the trap vector to determine where to go in the OS code.
• OS will not allow (or should not)
  • Users to read each other’s keyboards
  • Users to access all memory or disk locations
I/O Controller

- **Control/Status Registers**
  - CPU tells device what to do -- write to control register
  - CPU checks whether task is done -- read status register

**Data Registers**
- CPU transfers data to/from device

---

**Graphics Controller**

- **Control/Status**
- **Output Data**
- **Electronics**

**Device electronics**
- Performs actual operation
  - Pixels to screen, bits to/from disk, characters from keyboard
Programming Interface

How are device registers identified?

- Memory-mapped vs. special instructions

How is timing of transfer managed?

- Asynchronous vs. synchronous

Who controls transfer?

- CPU (polling) vs. device (interrupts)
Memory-Mapped vs. I/O Instructions

Special Instructions
- designate opcode(s) for I/O
- register and operation encoded in instruction

Memory-mapped
- assign a memory address to each device register
- use data movement instructions (LD/ST) for control and data transfer
Which is better?

861 + 5  256  16

- What is the problem with having special instructions for IO?
- What happens if a new device is created?

257th

Memory mapped is much more flexible and expandable.

3-4K
Memory Mapped IO

• Idea is to place devices other than RAM chips at physical address locations.
• This way to access IO devices you use the same load and store instructions.
Design hardware and software to recognize certain addresses.

- From keyboard: $0x00000000$ to display
- From keyboard: $0xffff0000$ to display
- From keyboard: $0xffff0008$ to display
- From keyboard: $0xffff0010$ to display

Real Memory - RAM
• Devices on bus watch for their address
• But is there a new char to read?
• But is the display done with the last char?
Linux $\rightarrow$ File

Need I/O device status to coordinate

Real Memory - RAM

- 0x00000000
- 0xffff0000
- 0xffff0008
- 0xffff000c
- 0xffff0010
- 0xffff0014

DATA from keyboard
STATUS from keyboard
DATA to Display
STATUS from Display
Transfer Timing

I/O events generally happen much slower than CPU cycles.

**Synchronous**
- data supplied at a fixed, predictable rate
- CPU reads/writes every X cycles

**Asynchronous**
- data rate less predictable
- CPU must synchronize with device, so that it doesn’t miss data or write too quickly
Transfer Control

Who determines when the next data transfer occurs?

Polling
- CPU keeps checking status register until new data arrives OR device ready for next data
- “Are we there yet? Are we there yet? Are we there yet?”

Interrupts
- Device sends a special signal to CPU when new data arrives OR device ready for next data
- CPU can be performing other tasks instead of polling device.
- “Wake me when we get there.”
LC-3

Memory-mapped I/O (Table A.3)

<table>
<thead>
<tr>
<th>Location</th>
<th>I/O Register</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>xFE00</td>
<td>Keyboard Status Reg (KBSR)</td>
<td>Bit [15] is one when keyboard has received a new character.</td>
</tr>
<tr>
<td></td>
<td>KBCR</td>
<td></td>
</tr>
<tr>
<td>xFE02</td>
<td>Keyboard Data Reg (KBDR)</td>
<td>Bits [7:0] contain the last character typed on keyboard.</td>
</tr>
<tr>
<td>xFE04</td>
<td>Display Status Register (DSR)</td>
<td>Bit [15] is one when device ready to display another char on screen.</td>
</tr>
<tr>
<td>xFE06</td>
<td>Display Data Register (DDR)</td>
<td>Character written to bits [7:0] will be displayed on screen.</td>
</tr>
</tbody>
</table>

Asynchronous devices
- synchronized through status registers

Polling and Interrupts

- the details of interrupts is in chapter 10
Input from Keyboard

When a character is typed:
- its ASCII code is placed in bits [7:0] of KBDR (bits [15:8] are always zero)
- the “ready bit” (KBSR[15]) is set to one
- keyboard is disabled -- any typed characters will be ignored

When KBDR is read:
- KBSR[15] is set to zero
- keyboard is enabled
Basic Input Routine

```
POLL LDI R0, KBSRPtr
BRzp POLL
LDI R0, KBDRPtr

KBSRPtr .FILL xFE00
KBDRPtr .FILL xFE02

spinning beach ball
```
Output to Monitor

When Monitor is ready to display another character:
- the “ready bit” (DSR[15]) is set to one

When data is written to Display Data Register:
- DSR[15] is set to zero
- character in DDR[7:0] is displayed
- any other character data written to DDR is ignored (while DSR[15] is zero)
Basic Output Routine

- **Polling**
  - screen ready?
    - NO
      - Polling
    - YES
      - write character

- Code:
  ```
  POLL LDI R1, DSRPtr
  BRzp POLL
  STI R0, DDRPtr
  ...
  DSRPtr .FILL xFE04
  DDRPtr .FILL xFE06
  ```
Keyboard Echo Routine

Usually, input character is also printed to screen.
- User gets feedback on character typed and knows it's ok to type the next character.

<table>
<thead>
<tr>
<th>Poll1</th>
<th>LDI R0, KBSRPtr</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BRzp POLL1</td>
</tr>
<tr>
<td>POLL2</td>
<td>LDI R1, DSRPtr</td>
</tr>
<tr>
<td></td>
<td>BRzp POLL2</td>
</tr>
<tr>
<td></td>
<td>STI R0, DDRPtr</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>KBSRPtr</td>
<td>.FILL xFE00</td>
</tr>
<tr>
<td>KBDRP</td>
<td>.FILL xFE02</td>
</tr>
<tr>
<td>DSRP</td>
<td>.FILL xFE04</td>
</tr>
<tr>
<td>DDRP</td>
<td>.FILL xFE06</td>
</tr>
</tbody>
</table>
Polling

- How much time is spent spinning?
- A PUTC or GETC is less than 10 instructions, or ~10ns on a modern processor
- Mechanical devices take milliseconds
- Almost all time is spent spinning
  - Would be nice to do useful work while waiting
  - Periodically poll devices and send characters when ready
Polling I/O

- The OS must check regularly (poll) for ready devices
  - Perhaps once a millisecond
- If ready, then OS services device
  - Keyboard: transfer character and put on queue
  - Display: transmit character to the graphics HW
Polling I/O

Problems:

- How often to poll?
- How does the OS code get run?
- What happens to the user program?

Is there a better solution?
Exceptions and the OS

STOP!!! Do this!!!
Interrupts

“External condition related to IO devices”

- Have device tell OS/CPU it is ready
- Requires hardware to support.
- OS/CPU can then interrupt what it is doing to:
  - Determine what device wants service
  - Determine what service it wants
  - Perform or start the service
  - Go back to what OS/CPU was doing
Examples of Interrupts

- Disk drive at sector/track position (old days)
- Mouse moved
- Keyboard key pressed
- Printer needs data
- Video card wants memory access
- Network sending or receiving
- USB scanner has data
Interrupt Properties

- They arrive asynchronously
- Can’t communicate with a running program (no args or return values)
- They are associated with various priorities
- You want to handle them soon (interrupt latency)
- (usually) want to resume program
Trap

“Internal conditions related to instruction stream”

Trap Examples:

- “TRAP” instruction
- Illegal instruction
- Arithmetic overflow
- Divide by zero
- “LD” from illegal/protected memory address
Trap Properties

- They arrive synchronously
- Get trap in same place if you re-run program
- They are associated with various priorities
- Must handle immediately
- Want to resume program (usually)
Exceptions

“Mechanism used to handle both interrupts and traps”

- HW handles initial reaction
- Then invokes SW called an “Exception Handler” to take care of the interrupt/trap
LC-3 Exceptions

- LC-3 only has the concept of IO related exceptions, interrupts.
- Does not have any mechanisms to deal with illegal instructions, or arithmetic overflow.
- Remember the LC-3 is just a learning aid, not real hardware.
LC-3 Interrupt-Driven I/O

External device can:
(1) Force currently executing program to stop;
(2) Have the processor satisfy the device’s needs; and
(3) Resume the stopped program as if nothing happened.

Why do interrupts?
– Polling consumes a lot of cycles, especially for rare events – these cycles can be used for more computation.
– Example: Process previous input while collecting current input. (See Example 8.1 in text.)
To implement an interrupt mechanism, we need:

- A way for the I/O device to **signal** the CPU that an interesting event has occurred.
- A way for the CPU to **test** whether the **interrupt signal is set** and whether its **priority is higher** than the current program.

**Generating Signal**

- Software sets "interrupt enable" bit in device register.
- When ready bit is set and IE bit is set, interrupt is signaled.
Priority

Every instruction executes at a stated level of urgency.

LC-3: 8 priority levels (PL0-PL7)

- Example:
  - Payroll program runs at PL0.
  - Nuclear power correction program runs at PL7.

- It’s OK for PL6 device to interrupt PL0 program, but not the other way around.

Priority encoder selects highest-priority device, compares to current processor priority level, and generates interrupt signal if appropriate.
Testing for Interrupt Signal

- CPU looks at signal between STORE and FETCH phases.
- If not set, continues with next instruction.
- If set, transfers control to interrupt service routine.

Diagram:
- Transfer to ISR
- interrupt signal?
  - NO
  - F
  - D
  - EA
  - OP
  - EX
  - S
- YES
Operating Systems

- The “program” that launches the user programs and deals with exceptions.
- How does the operating system deal with such things as simultaneous exceptions?
- What happens on machines that have many users “on” at once?
Multiple Exceptions

Problem: How about multiple simultaneous exceptions?

Solution: Have priorities in HW / SW

- Handle highest-priority exception first
- Equal priority exceptions handled arbitrarily
- Give higher priority
  - more serious (e.g., power failing)
  - can’t wait long (e.g., rotating disk)
How about exceptions during exception handling?

Option #1: Make it wait until done with first exception
  • May be a bad idea for higher priority exception

Option #2: Make exception handler re-entrant
  • Make able to handle multiple active calls
  • Allow higher-priority exceptions to bypass lower-priority exception currently being serviced