EE 3170 Microcontroller Applications

Lecture 8: Instruction Subset & Machine Language:
A Brief Tour of the 68HC11 Instruction Set.
- Miller §2.4 & §5.2- §5.3 & Appendix A

Based on slides for ECE3170 by Profs. Davis, Kieckhafer, Tan, and Cischke

Lecture Overview

- Expanding your repertoire of instructions
  - Instructions for simple arithmetic, logical, and shift operations
  - Condition codes & branch instructions

- A summary of 68HC11 instruction set
  - Getting very familiar with the format of the HC11 Reference Manual

What Logical Operations Can We Program?

- The 68HC11, like most processors, supports the basic logical operations of
  - NOT
  - AND
  - OR

- These instructions are applied bit-by-bit or bitwise.
  - For example if regA and regB are ANDed,
    - bits 7 of each register are ANDed,
    - bits 6 of each register are ANDed, yada yada yada

What Logical Operations Can We Program?

- The 68HC11 performs most of the basic logical operations you learned in 2173.
  - NOT output complement of input
  - AND output 1 only if both inputs 1
  - OR output 1 if either input 1
  - EOR output 1 if just 1 input 1 (XOR)

They are applied bit-by-bit (bitwise) to all bits of a byte.
For example if regA and regB are ANDed,
- bits 7 of each register are ANDed.
- bits 6 of each register are ANDed, yada yada yada
What are the HC11 Instructions for Logical AND Operations?

- **AND**
  - `anda #w` AND register A with number w
  - `anda u` AND reg. A with value in loc. u
  - `andb #w` AND reg. B with number w
  - `andb u` AND reg. B with value in loc. u

What are the HC11 Instructions for Logical OR/ EOR Operations?

- **OR**
  - `oraa`
  - `Orab`

- **EOR**
  - `eora`
  - `Eorb`

  As for AND these instructions can be used with #w, u

What are Practical Uses for HC11 Logical Instructions?

- **and** may be used to *isolate* a bit, usually for testing
  - Example: `anda #$04` isolates bit 2
    - bits are numbered from 0 at the right

- **and** may be used to *clear* a bit
  - Example: `anda #$F7` clears bit 3
    - the other bits are not affected

- **or** may be used to *set* a bit
  - Example: `oraa #$40` sets bit 6
    - the other bits are not affected

Some Hints on Mask Patterns

- Usually you want to work with one bit.
- Bits are numbered from 7 on the left to 0 on the right.
- We’ll show hex masks.

  - **Clearing** a bit takes a 0 for that bit position and 1s everywhere else.
    - $F$ for the unaffected nibble (half-byte)
    - $E$ to clear bit 0 or 4
    - $D$ to clear bit 1 or 5
    - $B$ to clear bit 2 or 6
    - $7$ to clear bit 3 or 7
Some Hints on Mask Patterns

- Setting a bit takes a 1 for that bit position and 0s elsewhere.
  - $0$ for the unaffected nibble
  - $1$ for bits 0 and 4
  - $2$ for bits 1 and 5
  - $4$ for bits 2 and 6
  - $8$ for bits 3 and 7

What can EOR be used for?

- EOR (exclusive OR) is used to toggle bits, that is to complement them.
- Often we want to repeatedly flip or toggle bits.
- This is used in a program to generate a square wave.
- But first we need to learn a little about the HC11 input/output structure (till chapter 6).

Complementing Bits

- Both NOT and EOR can be used to complement bits.
  - When would you use NOT?
  - When would you use EOR?

How Can We Shift Bits Within a Register?

- Treat a register or memory byte as a shift register
  - The HC11 includes the carry bit.
  - Other processors may not.
- Shift the data left or right 1-bit
- Options for the end bits
  - Rotate
  - Logical Shift
  - Arithmetic Shift
Rotate Shift Instructions

- **Rotate** instructions
  - Wrap the ends around
  - Treats the Carry bit as one bit of the register

Left:

```
  7 6 5 4 3 2 1 0
  C
```

Right:

```
  7 6 5 4 3 2 1 0
  C
```

Logarithmic Shift Instructions

- **Logical** shift instructions
  - Outgoing bit goes to carry
  - Incoming bit is zero

Logical Left Shift:

```
  C 7 6 5 4 3 2 1 0
```

Logical Right Shift:

```
  0 7 6 5 4 3 2 1 0 C
```

Arithmetic Shift Instructions

- **Arithmetic** shift instructions
  - Outgoing bit goes to carry
  - Incoming bit is arithmetically correct

Arithmetic Left Shift: shift in a zero

```
  C 7 6 5 4 3 2 1 0
```

Arithmetic Right Shift: copy the sign bit

```
  0 7 6 5 4 3 2 1 0 C
```

Shift Operations

- Rationale for arithmetic shifts
  - Left shift = multiply by 2
  - Right shift = divide by 2
  - Result is numerically correct for signed or unsigned, positive or negative

- Why shift through the Carry Bit?
  - Allows us to test bit values using branch-on-carry
  - Example— to test if a number is even
    - Right shift the number (C ← least significant bit)
    - follow with BCC or BCS instruction
    - BCC is Branch if Carry Clear (0)
    - BCS is Branch if Carry Set (1)
HC11 Shift Instructions

- For register A
  - asla arithmetic shift left
  - asra arithmetic shift right
  - Isla logical shift left
  - Isra logical shift right
  - rola rotate left
  - rora rotate right

- Registers B and D and memory have similar instructions.

Why Would We Want to Use These Shifts?

- Shifting into carry allows us to check carry bit.
  - We’ll see this later on.

- We can check individual bits by shifting into the carry.

How Can We Add and Subtract in HC11?

- We’ll be adding and subtracting in registers of limited size.
  - 8 or 16 bits
  - This limits the valid results we can calculate.

- We need to know
  - how to check if results are valid and
  - how to add numbers larger than 16 bits.

What are the HC11 8-bit Add and Subtract Instructions?

adda #w add w to regA
adda u add value in u to regA
suba #w subtract w from A
suba u subtract value in u from A

These are for signed and unsigned numbers. The result is in A in all cases.
What are the HC11 16-bit Add and Subtract Instructions?

- **addd #w**: add w to regD
- **addd u**: add value in u and u+1 to regD
- **subd #w**: subtract w from D
- **subd u**: subtract value in u and u+1 from D

These are for signed and unsigned numbers. The result is in D in all cases.

---

**Unsigned Addition/Subtraction**

- The numbers are added or subtracted.
- If the result takes more bits, the carry flag will be 1 showing overflow.
- Example 1: Add 64 and 64.
  
  \[
  \begin{array}{c}
  01000000 \\
  + 01000000 \\
  = 10000000
  \end{array}
  \]
  
  The result is 128 which is correct.

---

**Unsigned Addition/Subtraction**

- Example 2: Add 128 and 128
  
  \[
  \begin{array}{c}
  10000000 \\
  + 10000000 \\
  = 100000000
  \end{array}
  \]
  
  The register is 0 and the carry bit is 1, showing overflow.
  
  We can consider the carry bit as standing for the \(2^8\) or 256 position and consider the sum to be 256 plus 0 or 256.

---

**Addition**

- 2’s complement addition
  - Add two numbers together.
  - Simple and correct when in range
Subtraction

- 2's complement subtraction
  - Complement subtrahend and add

<table>
<thead>
<tr>
<th></th>
<th>(+5)</th>
<th>0 1 0 1</th>
<th>(-5)</th>
<th>1 0 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+2)</td>
<td>+ 0 0 1 0</td>
<td>(+2)</td>
<td>+ 0 0 1 0</td>
<td></td>
</tr>
<tr>
<td>(+7)</td>
<td>0 1 1 1</td>
<td>(-3)</td>
<td>1 1 0 1</td>
<td></td>
</tr>
</tbody>
</table>

Examples of 2's complement addition

<table>
<thead>
<tr>
<th></th>
<th>(+5)</th>
<th>0 1 0 1</th>
<th>(-5)</th>
<th>1 0 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-2)</td>
<td>+ 1 1 1 0</td>
<td>(-2)</td>
<td>+ 1 1 1 0</td>
<td></td>
</tr>
<tr>
<td>(+3)</td>
<td>1 0 0 1</td>
<td>(-7)</td>
<td>1 1 0 0</td>
<td></td>
</tr>
</tbody>
</table>

How Do We Know if the Result is Wrong?

- **Overflow** occurs when the result is too large to be represented in the register.

- Adding two positive or two negative numbers may result in overflow.

  - When this happens the sign bit of the sum is **negative** (1) for the sum of two **positive** numbers or **positive** (0) for the sum of two **negative** numbers.

What Does the CPU Do When Overflow Occurs?

- The CPU alerts us to overflow through condition codes.
- We can then take appropriate action.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>D</th>
<th>B</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit accumulators</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-bit accumulator</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Index register</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Index register</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack pointer</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program counter</td>
<td>PC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Condition code register</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Four HC11 Flags

<table>
<thead>
<tr>
<th>bit</th>
<th>name</th>
<th>meaning after add/subtract</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>carry</td>
<td>unsigned overflow</td>
</tr>
<tr>
<td>N</td>
<td>negative</td>
<td>result was negative</td>
</tr>
<tr>
<td>V</td>
<td>overflow</td>
<td>signed overflow</td>
</tr>
<tr>
<td>Z</td>
<td>zero</td>
<td>result was zero</td>
</tr>
</tbody>
</table>

How Does the CPU Know If Numbers are Signed or Not?

- It doesn’t.
- The CPU only understands bit patterns.
- We, as programmers, determine if numbers are signed or unsigned.
- The following examples show setting of condition codes and our interpretations of them.

How Do Condition Codes Work?

Example 1: Add $40 and $40.

```
01000000
+ 01000000
= 10000000
```

Sum is $80. C = 0 N = 1 V = 1 Z = 0

Unsigned interpretation— 64 + 64 = 128. valid
Signed interpretation— +64 ++64 = -128 invalid

Example 2: Add $80 and $80

```
10000000
+ 10000000
= 00000000
```

Sum is $00. C = 1 N = 0 V = 1 Z = 1

Unsigned interpretation — 128 + 128 = 0 plus a carry— invalid for single-byte addition
Signed interpretation— -128 + -128 = 0 invalid
How Do Condition Codes Work?

**Example 3: Add $FF and $01**

```
11111111
+ 00000001
=100000000
```

Sum is $00. C = 1  N = 0  V = 0  Z = 1

Unsigned interpretation — $255 + 1 = 0$ invalid for single-byte arithmetic

Signed interpretation — $-1 + +1 = 0$ valid

---

**Example 4: Add $80 and $01**

```
10000000
+ 00000001
= 10000001
```

Sum is $81. C = 0  N = 1  V = 0  Z = 0

Unsigned interpretation — $128 + 1 = 129$ valid

Signed interpretation — $-128 + +1 = -127$ valid

---

What Can We Do if Overflow Occurs?

- We want to recognize errors so we can correct them.
- Overflow is a major error.
- The HC11 provides branch instructions for this.
- We have previously seen bra, branch always, an unconditional branch instruction.
- Now we will see conditional branch instructions that let us test for overflow and branch (jump) to alternative instructions.

---

What Do Conditional Branch Instructions Do?

- Conditional branch instructions test one or more flags for a given condition.
  - If the condition is true, the program branches to a location given in the instruction.
  - If the condition is false, the program continues to the next instruction in sequence.
What are the HC11 Arithmetic Conditional Branch Instructions?

- **bcc** loc branch on carry clear (C=0)
- **bcs** loc branch on carry set (C=1)
- **bvc** loc branch on overflow clear (V=0)
- **bvs** loc branch on overflow set (V=1)
- **Bpl** loc branch on plus (N=0)
- **bmi** loc branch on minus (N=1)
- **beq** loc branch on equal (Z=0)
- **bne** loc branch on not equal (Z=1)

How Do We Use Conditional Branch Instructions in Adding?

- **Unsigned arithmetic**
  - `adda #$47`
  - `bcs Erroradd` (branch if carry set)

- **Signed arithmetic**
  - `adda #$47`
  - `bvs Error2add` (branch if overflow set)

How Do We Multiply?

- The HC11 only provides 8-bit unsigned multiplication.
- The **mul** instruction multiplies the values in A and B and puts the product in D.
- If we want to do signed multiplication, we
  - use **mul** as above on positive numbers
  - handle the sign of the product separately

How Do We Divide?

- The HC11 has no divide instructions.
  - `fdiv`
  - `Idiv`
  - `P208-209 (D/X)`
- Through algorithms shown in the text, we can divide.
- We will not cover division.
A Summary of 68HC11 Instruction Set

- Reading: §2.4 & Appendix A & § 5.2- § 5.3
  - §2.4 covers basic instructions related to A, B, X index reg., and CC reg.
  - Notations relevant to figure 2-41

- Instructions organized into categories
  - load, store, and transfer instructions
  - arithmetic instructions
  - logical instructions
  - bit testing and manipulation instructions
  - shifts and rotates
  - index-register and stack instructions
  - Branches

- Tables on the following slides: from Motorola’s M68HC11 Reference Manual.

Load, Store, and Transfer Instructions

- Accumulators A, B, and D can be loaded from or stored to memory.
- A and B can be transferred back and forth.
- A, B, and specified memory locations can be cleared (set to zero).
- A and B can be pushed to or pulled from the stack (more on this later).
- Double-precision accumulator D can be exchanged with the index registers X and Y

Load, Store, and Transfer Instructions

<table>
<thead>
<tr>
<th>Function</th>
<th>Mnemonic</th>
<th>IMM</th>
<th>DIR</th>
<th>EXT</th>
<th>INDX</th>
<th>IDY</th>
<th>INH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear Memory Byte</td>
<td>CLR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clear Accumulator A</td>
<td>CLRA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clear Accumulator B</td>
<td>CLRB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Accumulator A</td>
<td>LDAA</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Accumulator B</td>
<td>LDAB</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load Double Accumulator D</td>
<td>LDAD</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pop A from Stack</td>
<td>PIALA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pop B from Stack</td>
<td>PIBLB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Push A onto Stack</td>
<td>PSHLA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Push B onto Stack</td>
<td>PSHLB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Store Accumulator A</td>
<td>STA A</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Store Accumulator B</td>
<td>STAB</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Store Double Accumulator D</td>
<td>STAD</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transfer A to B</td>
<td>TAB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transfer A to CCR</td>
<td>TAR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transfer B to A</td>
<td>TRA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transfer CCR to A</td>
<td>TRA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exchange D with X</td>
<td>EXDX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exchange D with Y</td>
<td>EXDY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Logical Operations

- Accumulators A, can be logically ANDed, ORed, or exclusive-ORed with memory.
- A, B, and specified memory locations can be logically complemented (bit-wise).

<table>
<thead>
<tr>
<th>Function</th>
<th>Mnemonic</th>
<th>IMM</th>
<th>DR</th>
<th>EXT</th>
<th>INDX</th>
<th>INY</th>
<th>INH</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND A with Memory</td>
<td>ANDA</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>AND B with Memory</td>
<td>ANDB</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>AND C with Memory</td>
<td>ANDC</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>BITA with Memory</td>
<td>BITA</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>BITB with Memory</td>
<td>BITB</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>One's Complement A</td>
<td>COMA</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>One's Complement B</td>
<td>COMB</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>OR A with Memory (Exclusive)</td>
<td>ORA</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>OR B with Memory (Exclusive)</td>
<td>ORB</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>OR C with Memory (Exclusive)</td>
<td>ORC</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>OR D with Memory (Exclusive)</td>
<td>ORD</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Bit Testing and Manipulation

- Individual bits in memory can be set or cleared, or branches can be taken conditionally on their values.

<table>
<thead>
<tr>
<th>Function</th>
<th>Mnemonic</th>
<th>IMM</th>
<th>DR</th>
<th>EXT</th>
<th>INDX</th>
<th>INY</th>
<th>INH</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRSL Test A with Memory</td>
<td>BRSL</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>BRSL Test B with Memory</td>
<td>BRSL</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Clear Bits in Memory</td>
<td>BCLR</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Set Bits in Memory</td>
<td>BSSET</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Branch if Bits Clear</td>
<td>BRCLR</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Branch if Bits Set</td>
<td>BRSET</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Shifts and Rotates

- Accumulators A, B, D and specified memory locations can be shifted left or right by one bit position, either arithmetically or logically.
- A, B, D, and specified memory locations can be rotated left or right by one bit position (through the C bit of the condition code register).
- Note: There’s no difference between a logical and an arithmetic left shift.
Index-Register and Stack Instructions

- Index registers X and Y can be loaded from, stored to, or compared with memory, and pushed to or pulled from the stack.
- X, Y and SP can be incremented or decremented.
- Contents X or Y can be transferred to or from SP or exchanged with D.

### Branch Instructions

- Branches can be taken unconditionally or conditioned on one or more bits in the condition code register (no branch instruction affects CC bits).
  - A simple branch is conditioned on just one condition code bit.
  - A comparison branch is conditioned on the outcome of a preceding subtract or compare instruction and generally depends on more than one condition-code bit.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Microcode</th>
<th>REL</th>
<th>IMM</th>
<th>D</th>
<th>INDEX</th>
<th>IMMXY</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch 7 Carry Sign</td>
<td>BCE O</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch 6 Carry Sign</td>
<td>BCE O</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch 5 Equal Sign</td>
<td>BCE O</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch 4 Greater Than</td>
<td>BCE O</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch 3 Less Than</td>
<td>BCE O</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch 2 Equal to</td>
<td>BCE O</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch 1</td>
<td>BCE O</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch 0</td>
<td>BCE O</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch 12 Carry Sign</td>
<td>BCE O</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch 11 Carry Sign</td>
<td>BCE O</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch 10 Equal Sign</td>
<td>BCE O</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch 9 Greater Than</td>
<td>BCE O</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch 8 Less Than</td>
<td>BCE O</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch 7 Equal to</td>
<td>BCE O</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch 6 Equal to</td>
<td>BCE O</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch 5 Greater Than</td>
<td>BCE O</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch 4 Less Than</td>
<td>BCE O</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch 3 Equal to</td>
<td>BCE O</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch 2 Greater Than</td>
<td>BCE O</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch 1 Less Than</td>
<td>BCE O</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch 0</td>
<td>BCE O</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>For next time: Reading: §3.1–3.2, Miller</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Topic: introduction to assembly language</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>