General-purpose computers

• We’ve seen how to implement any algorithm as a digital circuit
• But we usually can’t implement a new digital circuits for each algorithm.
• If we could implement only ONE algorithm as a digital circuit, what should that algorithm be? We’d like to use that algorithm to solve as many problems as possible.
  – What makes a calculator useful?

• Remarkably, it’s possible to implement a single algorithm, yet still have that one algorithm carry out any algorithm. We call the implementation of such an algorithm a general-purpose computer.
  – Computer reads input (i.e. a program) that tells it how to carry out the other algorithms
  – Computer is able to execute general programs, so it’s able to carry out general algorithms
Stored-program computer

• The essence of what it means for a circuit to be a computer
  – Input instructions into the computer, just as data can be input to the computer
  – By inputting a new program, we can implement different algorithms
  – Computer manipulates the instructions in the same way it manipulates data

• We’re going to design a stored-program computer (processor) called the E100
  – Design the set of instructions that the computer can carry out
  – Implement a circuit (datapath + control unit) that can carry out any sequence of E100 instructions
  – Write algorithms as sequences of E100 instructions
Designing the set of instructions

• Key question in computer design
  – What are the instructions for the computer (what can you tell the computer to do)
  – If you pick the wrong set of instructions, you might not be able to express the desired algorithm using those instructions, i.e. the computer won’t be general purpose
  – E.g. if the only instruction the computer can do is increment, it’s going to be hard to tell it to compute A-B

• We’re going to design a small set of instructions that are simple, yet can be combined to compute arbitrary things
  – This is called the computer’s “instruction set”, or “instruction set architecture”, or ISA
Representing negative numbers (32-bit word)

• Bit 31 is worth -2,147,483,648 (instead of 2,147,483,648)

• What is the largest positive number you can represent in 32 bits?

• What is the largest negative number you can represent in 32 bits?

• What value does 1000 0000 0000 0000 0000 0000 0000 0001 represent?

• What value does 1111 1111 1111 1111 1111 1111 1111 1111 represent?

• E100 treats all numbers as signed

  — 32’hFFFFFFFF + 32’h00000003 =
E100 instruction set architecture

• Word is 32 bits
  – Data (i.e. variables) are 32 bits
  – Memory address is 32 bits. Only 16384 words on Cyclone IV FPGA, so only 14 bits of the address are used)

• Instructions and data are stored in memory
• An E100 instruction consists of 4 words: opcode, arg1, arg2, arg3
  – opcode specifies the operation to do at this step (e.g., add)
  – arg1, arg2, arg3 are the parameters for this operation (e.g., where to find the values to add, where to store the result)
• We store these 4 words in memory. Let IAR (instruction address register) be the address of the first word of the current instruction (the one being executed). The instruction is stored in mem[IAR] through mem[IAR+3]

  mem[ ] opcode
  mem[ ] arg1
  mem[ ] arg2
  mem[ ] arg3

• A processor executes an (infinite) loop of instructions
  – An instruction will typically perform some computation, and also determine the address of the next instruction to execute.
  – What should a typical instruction change IAR to?
E100 instructions

• HALT (opcode 0)
  – Tell the computer to stop executing instructions
  – First word (opcode) of the instruction has the value 0
  – Next three words of the instruction are ignored
    
    mem[IAR]  0
    mem[IAR+1]  0
    mem[IAR+2]  0
    mem[IAR+3]  0

• ADD (opcode 1)
  – Add two variables, store the result in another variable
    
    mem[IAR]  1
    mem[IAR+1]  address where to store the sum
    mem[IAR+2]  address of first addend
    mem[IAR+3]  address of second addend
Example E100 program

\[ \text{mem}[100] = \text{mem}[101] + \text{mem}[102] \]

\begin{align*}
\text{mem}[0] & \quad 1 \\
\text{mem}[1] & \quad 100 \\
\text{mem}[2] & \quad 101 \\
\text{mem}[3] & \quad 102 \\
\text{mem}[4] & \quad 0 \\
\text{mem}[5] & \quad 0 \\
\text{mem}[6] & \quad 0 \\
\text{mem}[7] & \quad 0 \\
\end{align*}

... 

\begin{align*}
\text{mem}[100] & \quad 0 \\
\text{mem}[101] & \quad 22 \\
\text{mem}[102] & \quad 33 \\
\end{align*}

• What happens when the E100 executes the first instruction?

• Note the difference between the address of the operands and the data of those operands
  – Addresses in the instruction specify where in memory the operands are
  – The actual data being added are stored in the memory word pointed to by an address
Other arithmetic instructions in the E100 ISA

- **SUB (opcode 2) (subtract)**
  \[ \text{mem}[\text{arg1}] = \text{mem}[\text{arg2}] - \text{mem}[\text{arg3}] \]

- **MULT (opcode 3) (multiply)**
  \[ \text{mem}[\text{arg1}] = \text{mem}[\text{arg2}] \times \text{mem}[\text{arg3}] \]

- **DIV (opcode 4) (divide)**
  \[ \text{mem}[\text{arg1}] = \frac{\text{mem}[\text{arg2}]}{\text{mem}[\text{arg3}]} \]

- **CP (opcode 5) (copy)**
  \[ \text{mem}[\text{arg1}] = \text{mem}[\text{arg2}] \]
Are arithmetic instructions sufficient?

• What kinds of programs can we implement with arithmetic instructions?

• What kinds can we not implement?
Conditional branches

- **BE** (opcode 13) (branch if equal)
  
  \[
  \text{if (mem[arg2] == mem[arg3]) goto arg1}
  \]
  
  - All the arithmetic instructions incremented IAR by 4 as part of their execution.
  - BE sets IAR to the branch target (arg1) if the two variables are equal. If they’re not equal, BE increments IAR like the other instructions
  - A conditional “goto” statement
  - Note difference between address and data. mem[arg2] may be equal to mem[arg3], even if arg2 is not equal to arg3

- **BNE** (opcode 14) (branch if not equal)
  
  \[
  \text{if (mem[arg2] != mem[arg3]) goto arg1}
  \]

- **BLT** (opcode 15) (branch if less than)
  
  \[
  \text{if (mem[arg2] < mem[arg3]) goto arg1}
  \]
  
  - remember that E100 numbers are signed
  - e.g. FFFF is less than 0000
Implement difference via branch instructions

if (mem[100] < mem[101]) {
    mem[102] = mem[101] - mem[100];
} else {
    mem[102] = mem[100] - mem[101];
}

• How could you write this with if-goto?
Translate difference into E100 instructions
Simulating an initial memory image
General data structures

• What kind of data structures can NOT be manipulated via the current instruction set (arithmetic, branch)? Why?
Accessing arrays

• CPFA (opcode 11) (copy from array)
  \[ \text{mem}[\text{arg1}] = \text{mem}[\text{arg2} + \text{mem}[\text{arg3}]] \]

• E.g. \( x = \text{array}[i] \)
  – The variable \( i \) is stored in \( \text{mem}[101] \)
  – The variable \( x \) is stored in \( \text{mem}[100] \)
  – The array is stored in \( \text{mem}[200] \) and following
    \[
    \begin{align*}
    \text{mem}[100] & \quad \text{(x)} \\
    \text{mem}[101] & \quad \text{(i)} \\
    \text{mem}[200] & \quad 1000 \quad \text{(array[0])} \\
    \text{mem}[201] & \quad 3000 \quad \text{(array[1])} \\
    \text{mem}[202] & \quad 5000 \quad \text{(array[2])} \\
    \text{mem}[203] & \quad 8000 \quad \text{(array[3])}
    \end{align*}
    \]

• CPFA 100 200 101
  \[
  \begin{align*}
  \text{arg1} &= 100 \\
  \text{arg2} &= 200 \\
  \text{arg3} &= 101
  \end{align*}
  \]
• Address of array element being accessed is:
• CPTA (opcode 12) (copy to array)
  \[ \text{mem[arg2+mem[arg3]]} = \text{mem[arg1]} \]

• e.g. \( \text{array[i]} = x \)
Implementing function calls

• When using a function, when does the next instruction to execute not follow sequentially after the prior instruction?

```c++
main() {
    i = 0;
    func(i);

    i = 1;
    func(i);

    i = 2;
}

func(int i) {
    cout << i << endl;
    return;
}
```
• Branch instructions go to a constant address, i.e. the target address is specified in the instruction
Calling and returning from functions

- **RET (opcode 17) (return)**
  
  \[ \text{IAR} = \text{mem}[\text{arg1}] \]

  - E.g., if \( \text{mem}[100] = 4 \), then what will executing “RET 100” do?

- **CALL (opcode 16)**

  \[ \text{mem}[\text{arg2}] = \text{address of the instruction after CALL instruction. Why?} \]

  \[ \text{IAR} = \text{arg1} \]
Example of call/return

mem[0] 16
mem[1] 100
mem[2] 120
mem[3] 0

mem[4] 1
mem[5] 200
mem[6] 201
mem[7] 202

mem[100] 2
mem[101] 300
mem[102] 301
mem[103] 302

mem[104] 17
mem[105] 120
mem[106] 0
mem[107] 0

mem[120] 0
Implementing the E100

• We know how to implement any algorithm as a digital circuit
  – Datapath
  – Control unit (FSM)
• Overview
  – We’re designing a digital circuit that implements an E100 ISA
  – This digital circuit will execute E100 instructions stored in memory (i.e. an E100 program)

• Steps in executing an instruction
  – Fetch the instruction from memory
  – Decide what to do, based on the opcode for the instruction
  – Execute the instruction
Implementing E100’s ADD instruction

- C++ version of algorithm
  - Remember what ADD does:
    
    \[
    \begin{align*}
    \text{mem[IAR]} & \quad 1 \\
    \text{mem[IAR+1]} & \quad \text{arg1} \\
    \text{mem[IAR+2]} & \quad \text{arg2} \\
    \text{mem[IAR+3]} & \quad \text{arg3} \\
    \end{align*}
    \]

    \[
    \text{mem[arg1]} = \text{mem[arg2]} + \text{mem[arg3]} \\
    \text{IAR} = \text{IAR} + 4
    \]
• Fetch

• Decode

• Execute
Datapath for E100
<table>
<thead>
<tr>
<th>reset</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>opcode-out</td>
</tr>
<tr>
<td></td>
<td>equal-out</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>fetch1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>next-state</td>
<td>pc-write</td>
</tr>
<tr>
<td></td>
<td>pc-drive</td>
</tr>
<tr>
<td></td>
<td>plus1-drive</td>
</tr>
<tr>
<td></td>
<td>op1-write</td>
</tr>
<tr>
<td></td>
<td>op2-write</td>
</tr>
<tr>
<td></td>
<td>add-drive</td>
</tr>
<tr>
<td></td>
<td>opcode-write</td>
</tr>
<tr>
<td></td>
<td>arg1-write</td>
</tr>
<tr>
<td></td>
<td>arg1-drive</td>
</tr>
<tr>
<td></td>
<td>arg2-write</td>
</tr>
<tr>
<td></td>
<td>arg2-drive</td>
</tr>
<tr>
<td></td>
<td>arg3-write</td>
</tr>
<tr>
<td></td>
<td>arg3-drive</td>
</tr>
<tr>
<td></td>
<td>address-write</td>
</tr>
<tr>
<td></td>
<td>mem-write</td>
</tr>
<tr>
<td></td>
<td>mem-drive</td>
</tr>
<tr>
<td>state</td>
<td>opcode_out</td>
</tr>
<tr>
<td>-------</td>
<td>------------</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
always @* begin
    // default values for control signals
    pc_write = 1'b0;
    pc_drive = 1'b0;
    plus1_drive = 1'b0;
    op1_write = 1'b0;
    op2_write = 1'b0;
    add_drive = 1'b0;
    opcode_write = 1'b0;
    arg1_write = 1'b0;
    arg1_drive = 1'b0;
    arg2_write = 1'b0;
    arg2_drive = 1'b0;
    arg3_write = 1'b0;
    arg3_drive = 1'b0;
    address_write = 1'b0;
    memory_write = 1'b0;
    memory_drive = 1'b0;
    next_state = state_reset;
case (state)

    state_reset: begin
        next_state = state_fetch1;
    end

    // fetch the current instruction

    state_fetch1: begin
        // copy pc to address
        pc_drive = 1'b1;
        address_write = 1'b1;
        next_state = state_fetch2;
    end

    state_fetch2: begin
        // read opcode from memory
        memory_drive = 1'b1;
        opcode_write = 1'b1;
        next_state = state_fetch3;
    end
state_fetch3: begin
    // increment pc; copy new value to address
    plus1_drive = 'b1;
    pc_write = 'b1;
    address_write = 'b1;
    next_state = state_fetch4;
end

state_fetch4: begin
    // read arg1 from memory
    memory_drive = 'b1;
    arg1_write = 'b1;
    next_state = state_fetch5;
end

state_fetch5: begin
    // increment pc; copy new value to address
    plus1_drive = 'b1;
    pc_write = 'b1;
    address_write = 'b1;
    next_state = state_fetch6;
end
state_fetch6: begin
    // read arg2 from memory
    memory_drive = 1'b1;
    arg2_write = 1'b1;
    next_state = state_fetch7;
end

state_fetch7: begin
    // increment pc; copy new value to address
    plus1_drive = 1'b1;
    pc_write = 1'b1;
    address_write = 1'b1;
    next_state = state_fetch8;
end

state_fetch8: begin
    // read arg3 from memory
    memory_drive = 1'b1;
    arg3_write = 1'b1;
    next_state = state_decode;
end
// decode the current instruction

state_decode: begin
    // transfer address of (probable) next instruction to pc
    plus1_drive = 1'b1;
    pc_write = 1'b1;

    // choose next state, based on opcode
    if (opcode_out == E100_ADD) begin
        next_state = state_add1;
    end else if (opcode_out == E100_BE) begin
        next_state = state_be1;
    end

    end
end
// execute add instruction

state_add1: begin
    // transfer arg2 to address
    arg2_drive = 1'b1;
    address_write = 1'b1;
    next_state = state_add2;
end

state_add2: begin
    // transfer mem[arg2] to op1
    memory_drive = 1'b1;
    op1_write = 1'b1;
    next_state = state_add3;
end

state_add3: begin
    // transfer arg3 to address
    arg3_drive = 1'b1;
    address_write = 1'b1;
    next_state = state_add4;
end
state_add4: begin
  // transfer mem[arg3] to op2
  memory_drive = 1'b1;
  op2_write = 1'b1;
  next_state = state_add5;
end

state_add5: begin
  // transfer arg1 to address
  arg1_drive = 1'b1;
  address_write = 1'b1;
  next_state = state_add6;
end

state_add6: begin
  // write op1 + op2 to mem[arg1]
  add_drive = 1'b1;
  memory_write = 1'b1;
  next_state = state_fetch1;
end
// execute be instruction

state_be1: begin
    // transfer arg2 to address
    arg2_drive = 1'b1;
    address_write = 1'b1;
    next_state = state_be2;
end

state_be2: begin
    // transfer mem[arg2] to op1
    memory_drive = 1'b1;
    op1_write = 1'b1;
    next_state = state_be3;
end

state_be3: begin
    // transfer arg3 to address
    arg3_drive = 1'b1;
    address_write = 1'b1;
    next_state = state_be4;
end
state_be4: begin
  // transfer mem[arg3] to op2
  memory_drive = 1'b1;
  op2_write = 1'b1;
  next_state = state_be5;
end

state_be5: begin
  // if (op1 == op2) take branch
  if (equal_out == 1'b1) begin
    next_state = state_be6;
  end else begin
    next_state = state_fetch1;
  end
end

state_be6: begin
  // transfer arg1 to pc
  arg1_drive = 1'b1;
  pc_write = 1'b1;
  next_state = state_fetch1;
end
endcase
Writing programs for the E100

• Recall the program to compute the difference between mem[20] and mem[21]

• Pseudocode:

```plaintext
if (mem[20] < mem[21]) goto LESS
goto END

LESS

END
halt
```
Difference algorithm (machine code)

<table>
<thead>
<tr>
<th>mem[0]</th>
<th>15 (BLT)</th>
<th>mem[12]</th>
<th>2 (SUB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem[4]</td>
<td>2 (SUB)</td>
<td>mem[16]</td>
<td>0 (HALT)</td>
</tr>
<tr>
<td>mem[6]</td>
<td>20</td>
<td>mem[18]</td>
<td>0</td>
</tr>
<tr>
<td>mem[10]</td>
<td>0</td>
<td>mem[22]</td>
<td>0</td>
</tr>
<tr>
<td>mem[11]</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
• What if I wanted to add a line of code before this program, e.g. `mem[20] = mem[20] + 1`?

• How can we make programs easier to write and modify?
Assembler

- Program that translates E100 assembly-language file into initial memory image
  - Translates symbolic addresses into numeric addresses
  - Provides other features to make it a little easier to write programs for the E100 ISA
- Assembly language format
  
  [label]  opcode  arg1  arg2  arg3

- Fields are separated by white space (spaces or tabs)
- Label gives a name to the (first) address for this line of code
  - Label is optional
  - If label is absent, then there must be white space before opcode
    (otherwise opcode will look like a label)
- arg1, arg2, arg3 can be decimal number, hexadecimal number (prefix with 0x), or label
- Comments marked by // (rest of line is ignored)
- Blank lines ignored
- Unspecified locations filled in with 0
Difference algorithm in assembly language

```
blt less x y
sub result x y
be end 0 0
less    sub result y x
end     halt
```

- How to initialize variables \((x, y)\)?