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 \(-2^{31}\) (instead of \(2^{31}\))
- 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 \(1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001\) represent?
- What value does \(1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\) 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

mem[100] = mem[101] + mem[102]
mem[0] 1
mem[1] 100
mem[2] 101
mem[3] 102
mem[4] 0
mem[5] 0
mem[6] 0
mem[7] 0
...
mem[100] 0
mem[101] 22
mem[102] 33

• 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)**
  \[ \text{mem}[\text{arg1}] = \text{mem}[\text{arg2}] / \text{mem}[\text{arg3}] \]

- **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)
  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)
  if (mem[arg2] != mem[arg3]) goto arg1

• BLT (opcode 15) (branch if less than)
  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)
  \[ \text{mem[arg1]} = \text{mem[arg2 + mem[arg3]]} \]
- 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
    \[
    \begin{align*}
    \text{mem[100]} &\quad \text{mem[101]} \\
    \text{mem[200]} &\quad 1000 \quad \text{(array[0])} \\
    \text{mem[201]} &\quad 3000 \quad \text{(array[1])} \\
    \text{mem[202]} &\quad 5000 \quad \text{(array[2])} \\
    \text{mem[203]} &\quad 8000 \quad \text{(array[3])}
    \end{align*}
    \]
- CPFA 100 200 101
  \[
  \begin{align*}
  \text{arg1} &= 100 \\
  \text{arg2} &= 200 \\
  \text{arg3} &= 101
  \end{align*}
  \]
- Address of array element being accessed is:
Implementing function calls

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

```cpp
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:

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem[IAR]</td>
<td>1</td>
</tr>
<tr>
<td>mem[IAR+1]</td>
<td>arg1</td>
</tr>
<tr>
<td>mem[IAR+2]</td>
<td>arg2</td>
</tr>
<tr>
<td>mem[IAR+3]</td>
<td>arg3</td>
</tr>
</tbody>
</table>

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

• Fetch

• Decode

• Execute
Datapath for E100
<table>
<thead>
<tr>
<th>state</th>
<th>opcode_out</th>
<th>equal_out</th>
<th>next_state</th>
<th>pc_write</th>
<th>pc_drive</th>
<th>plus1_drive</th>
<th>op1_write</th>
<th>op2_write</th>
<th>add_drive</th>
<th>opcode_write</th>
<th>arg1_write</th>
<th>arg1_drive</th>
<th>arg2_write</th>
<th>arg2_drive</th>
<th>arg3_write</th>
<th>arg3_drive</th>
<th>address_write</th>
<th>mem_write</th>
<th>mem_drive</th>
<th>reset</th>
<th>fetch1</th>
</tr>
</thead>
<tbody>
<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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Control unit for E100 (Verilog)

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;
end

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

end
state_fetch3: begin
    // increment pc; copy new value to address
    plus1_drive = 1'b1;
    pc_write = 1'b1;
    address_write = 1'b1;
    next_state = state_fetch4;
end

state_fetch4: begin
    // read arg1 from memory
    memory_drive = 1'b1;
    arg1_write = 1'b1;
    next_state = state_fetch5;
end

state_fetch5: begin
    // increment pc; copy new value to address
    plus1_drive = 1'b1;
    pc_write = 1'b1;
    address_write = 1'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

// 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

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
Writing programs for the E100

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

\[
\text{if (mem}[20] < \text{mem}[21]) \text{ goto LESS} \\
\text{mem}[22] = \text{mem}[20] - \text{mem}[21] \\
\text{goto END} \\
\text{LESS} \quad \text{mem}[22] = \text{mem}[21] - \text{mem}[20] \\
\text{END} \quad \text{halt}
\]
Difference algorithm (machine code)

mem[0] 15 (BLT)
mem[2] 20
mem[3] 21
mem[4] 2 (SUB)
mem[5] 22
mem[6] 20
mem[7] 21
mem[8] 13 (BE)
mem[9] mem[16] 0 (HALT)
mem[10] 0
mem[11] 0
mem[12] 2 (SUB)
mem[13] 22
mem[14] 21
mem[15] 20
mem[16] 0 (HALT)
mem[17] 0
mem[18] 0
mem[19] 0
mem[20] 50
mem[21] 60
mem[22] 0

- 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
  
  \[
  \text{[label]} \quad \text{opcode} \quad \text{arg1} \quad \text{arg2} \quad \text{arg3}
  \]

  - 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)?
Implement if-then-else in assembly language

if (x) {
    <then_clause>
} else {
    <else_clause>
}

becomes:

    if (!x) goto else_clause
    <then_clause>
    goto after_if
else_clause:    <else_clause>
after_if:        ...

Implementing loops in assembly language

• Count from 0 to 3
While loop

while (!end_condition) {
    <body of loop>
}

becomes:

loop:                       if (end_condition) goto end
<body of loop>
goto loop
end                             ...

Do-while loop

do {
    <body of loop>
} while (!end_condition)

becomes:

loop: <body of loop>
    if (!end_condition) goto loop
...

Find the maximum of mem[0] through mem[15]
Calling functions in assembly language

• Example: main program needs to compute difference between several pairs of numbers. Write a function to compute the difference between two numbers, and have the overall program call that function several times.
• What is the interface to this function? How do I use it?

• Calling the function

• Why are functions a good idea?

• Note the naming convention: Prefix all labels with name of file. Why is this a good idea?
Implementing algorithms in hardware vs. in software

• Any algorithm can be implemented in hardware or in software
  – E.g. diff, max, rot13
  – Compare these implementations

• What about the E100? We built a digital circuit that implemented the E100 ISA. Could we build a software program that implemented the E100 ISA?

Bit operations

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>AND(A,B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

• AND(0,X) =
• AND(1,X) =
Bit operations

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>OR(A,B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- OR(0,X) =
- OR(1,X) =

Bit operations

<table>
<thead>
<tr>
<th>A</th>
<th>NOT(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
E100 bit-manipulation instructions

• AND (opcode 6) (bitwise and)
  – mem[arg1] = (mem[arg2] & mem[arg3])
  – E.g. what is 9 & 10?

• OR (opcode 7) (bitwise or)
  – mem[arg1] = (mem[arg2] | mem[arg3])
  – E.g. what is 9 | 10?

• NOT (opcode 8) (bitwise negation)
  – mem[arg1] = ~(mem[arg2]) (arg3 is unused)
  – E.g. what is ~9?

• SL (opcode 9) (shift left)
  – E.g. what is 9 << 2?

  – SL shifts 0s into the least-significant bit(s)

• SR (opcode 10) (shift right)
  – E.g. what is 9 >> 2?

  – SR shifts 0s into the most-significant bit(s)
Programming in assembly language: manipulating bit fields

• Often useful to manipulate a range of bits in a word

• E.g. X is a 32-bit number. How to tell it’s even or odd?

• How to extract bits 7-4 of a word?

• How to set bit 0 of a word to 1 (leaving the rest of the word unchanged)?

• How to set bits 7-4 of a word to 1111 (leaving the rest of the word unchanged)?

• How to clear bits 7-4 of a word to 0000 (leaving the rest of the word unchanged)?
Lookup table

• Implement a program in C++ that maps one set of numbers to another set of numbers
  0 maps to 100
  1 maps to 59
  2 maps to 83
  3 maps to 92
  etc.

Implement lookup table in assembly language
A lookup table of arrays

- What if you wanted to map a number to a variable-length list of numbers?
  - 0 maps to \{100, 102, 104, 106, 0\}
  - 1 maps to \{59, 57, 0\}
  - 2 maps to \{83, 0\}
  - 3 maps to \{92, 90, 99, 0\}
  - etc.
Input/output on the E100

• So far, all “input” has been entered by an initial memory image, and all “output” has been produced by storing values in memory.

• DE2 includes many I/O devices
  – Input: switches, microphone, PS/2 keyboard, USB mouse, secure digital card
  – Output: LEDs, 7-segment LEDs, LCD, speaker
  – Input/output: SDRAM, VGA, serial port

• DE2 or E100 provides controllers for each of the complex I/O devices (LCD, VGA, PS/2, USB, speaker, microphone, SDRAM, SD card, serial port)

• Similar to commercial computers
  – Graphics cards (e.g., NVIDIA, ATI)
  – Sound cards (e.g., SoundBlaster)

  – Program issues commands to the controller to cause them to do something. E.g. program tells graphics card to clear the screen

I/O registers

• E100 programs communicate with an I/O controller through I/O registers
  – An E100 program can read or write a register by referring to the memory address assigned to that register
  – This is called “mapping” the I/O register to a memory address, so we call these “memory-mapped” registers
  – Caveat: I/O registers on the E100 can be read or written, but not both

• E.g., 0x80000000 is assigned to SW

• \[ \text{cp a 0x80000000} \] // copies SW to a

• E.g., 0x80000004 is assigned to HEX7-HEX4

\[ \text{cp 0x80000004 num15} \] // displays the value of num15
\[ \text{15} \] // num15 to HEX7-HEX4

• In this case, the I/O controller is just hexdigit, which converts the value to a hex digit displayed on the 7-segment LEDs
Communicating a series of numbers

• What problems did you encounter when trying to understand the series of numbers I was communicating to you?

• What problems did I encounter when trying to communicate numbers to you?

Communication protocols

• Need a protocol to send sequence of commands to I/O controller
• Signals for the protocol
  – command\_parameters: the data that is being sent from the E100 to the I/O controller to describe the command
  – command: E100 sets this to tell the I/O controller to execute the command.
  – response\_parameters: the data being sent from I/O controller to E100 in response to the requested command.
  – response: I/O controller sets this when it has executed the command and is ready with the response
Protocol for sending command to an I/O controller

- Start with command==0, response==0 (system is idle)
- E100 simulator (ase100) simulates the E100’s I/O controllers and I/O devices
Other protocols

• E100 bus uses a different type of “protocol”
  – Goal of bus protocol is to make sure only one component is driving the bus at a time
  – Who carries out this protocol?