

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 (16 bit word)

Bit 15 is worth -32768 (instead of 32768)

What is the largest positive number you can represent in 16 bits?

What is the largest negative number you can represent in 16 bits?

What value does 1000 0000 0000 0001 represent?

What value does 1111 1111 1111 1111 represent?

E100 treats all numbers as signed

- $16'hFFFF + 16'h0003 =$

E100 instruction set architecture

Word is 16 bits

- data (i.e. variables) are 16 bits
- memory address is 16 bits. Only 16384 words on Cyclone II FPGA, so only 14 bits of the address are used)

Instructions and data are stored in memory

Each instruction is 4 words

- those 4 words describe what the instruction is to do

E100 execution

- starts by executing the instruction stored in memory[0], memory[1], memory[2], memory[3]
- typically then executes the instruction stored in memory[4], memory[5], memory[6], memory[7]

Program counter (PC) contains the starting address of the current instruction

- next instruction to be executed is usually the next one, i.e.
PC =

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 [PC]      0
mem [PC+1]    0
mem [PC+2]    0
mem [PC+3]    0
```

ADD (opcode 1)

- add two variables, store the result in another variable
- what do I need to tell the computer for it to carry out this instruction?

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

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)

- $\text{mem}[\text{addr0}] = \text{mem}[\text{addr1}] - \text{mem}[\text{addr2}]$

MULT (opcode 3) (multiply)

- $\text{mem}[\text{addr0}] = \text{mem}[\text{addr1}] * \text{mem}[\text{addr2}]$

DIV (opcode 4) (divide)

- $\text{mem}[\text{addr0}] = \text{mem}[\text{addr1}] / \text{mem}[\text{addr2}]$

CP (opcode 5) (copy)

- $\text{mem}[\text{addr0}] = \text{mem}[\text{addr1}]$

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 ($\text{mem}[\text{addr1}] == \text{mem}[\text{addr2}]$) goto addr0
- all the arithmetic instructions incremented PC by 4 as part of their execution. BE sets PC to the **branch target** (addr0) if the two variables are equal. If they're not equal, BE increments PC like the other instructions
- “go to”
- note difference between address and data: $\text{mem}[\text{addr1}]$ may be equal to $\text{mem}[\text{addr2}]$, even if addr1 is not equal to addr2

BNE (opcode 14) (branch if not equal)

- if ($\text{mem}[\text{addr1}] \neq \text{mem}[\text{addr2}]$) goto addr0

BLT (opcode 15) (branch if less than)

- if ($\text{mem}[\text{addr1}] < \text{mem}[\text{addr2}]$) goto addr0
- 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?

CPFA and CPTA

CPFA (opcode 11) (copy from array)

- $\text{mem}[\text{addr0}] = \text{mem}[\text{addr1} + \text{mem}[\text{addr2}]]$

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

$\text{mem}[100]$		(x)
$\text{mem}[101]$		(i)
$\text{mem}[200]$	1000	$(\text{array}[0])$
$\text{mem}[201]$	3000	$(\text{array}[1])$
$\text{mem}[202]$	5000	$(\text{array}[2])$
$\text{mem}[203]$	8000	$(\text{array}[3])$

CPFA 100 200 101

- $\text{addr0} = 100$
- $\text{addr1} = 200$
- $\text{addr2} = 101$

- address of array element being accessed is:

CPTA (opcode 12) (copy to array)

- $\text{mem}[\text{addr1} + \text{mem}[\text{addr2}]] = \text{mem}[\text{addr0}]$

e.g. $\text{array}[i] = x$

Implementing function calls

What transfers of control occur when calling a function?

```
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

CALL and RET

RET (opcode 17) (return)

- PC = mem[addr0]

mem[100] 4

RET 100

- PC gets the value 4

CALL (opcode 16)

- mem[addr1] = address of the instruction **after** CALL instruction. Why?
- PC = addr0

Example

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:

```
mem[PC]      1
mem[PC+1]    addr0
mem[PC+2]    addr1
mem[PC+3]    addr2
```

```
mem[addr0] = mem[addr1] + mem[addr2]
PC = PC + 4
```

Datapath for E100

Fetch

Decode

Execute

state			next_state																	

Control unit for E100 (Verilog)

```
always @* begin
    // default values for control signals
    pc_write = 1'b0;
    pc_drive = 1'b0;
    alu_op1_write = 1'b0;
    alu_op2_write = 1'b0;
    alu_add_drive = 1'b0;
    plus1_drive = 1'b0;
    opcode_write = 1'b0;
    addr0_write = 1'b0;
    addr0_drive = 1'b0;
    addr1_write = 1'b0;
    addr1_drive = 1'b0;
    addr2_write = 1'b0;
    addr2_drive = 1'b0;
    address_write = 1'b0;
    memory_write = 1'b0;
    memory_drive = 1'b0;
    next_state = state_halt;

    if (state == state_reset) begin
        next_state = state_fetch1;
    end
end
```

```
    // fetch the current instruction

end else if (state == state_fetch1) begin
    // copy pc to address
    pc_drive = 1'b1;
    address_write = 1'b1;
    next_state = state_fetch2;

end else if (state == state_fetch2) begin
    // read opcode from memory
    memory_drive = 1'b1;
    opcode_write = 1'b1;
    next_state = state_fetch3;

end else if (state == 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 else if (state == state_fetch4) begin
    // read addr0 from memory
    memory_drive = 1'b1;
    addr0_write = 1'b1;
    next_state = state_fetch5;

end else if (state == 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
```

```

end else if (state == state_fetch6) begin
    // read addr1 from memory
    memory_drive = 1'b1;
    addr1_write = 1'b1;
    next_state = state_fetch7;

end else if (state == 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 else if (state == state_fetch8) begin
    // read addr2 from memory
    memory_drive = 1'b1;
    addr2_write = 1'b1;
    next_state = state_decode;

// decode the current instruction
end else if (state == 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_bel;
    end
end

```

```

// execute add instruction

end else if (state == state_add1) begin
    // transfer addr1 to address
    addr1_drive = 1'b1;
    address_write = 1'b1;
    next_state = state_add2;

end else if (state == state_add2) begin
    // transfer mem[addr1] to alu_op1
    memory_drive = 1'b1;
    alu_op1_write = 1'b1;
    next_state = state_add3;

end else if (state == state_add3) begin
    // transfer addr2 to address
    addr2_drive = 1'b1;
    address_write = 1'b1;
    next_state = state_add4;

end else if (state == state_add4) begin
    // transfer mem[addr2] to alu_op2
    memory_drive = 1'b1;
    alu_op2_write = 1'b1;
    next_state = state_add5;

end else if (state == state_add5) begin
    // transfer addr0 to address
    addr0_drive = 1'b1;
    address_write = 1'b1;
    next_state = state_add6;

```

```

end else if (state == state_add6) begin
    // write alu_op1 + alu_op2 to mem[addr0]
    alu_add_drive = 1'b1;
    memory_write = 1'b1;
    next_state = state_fetch1;
end

```

```

// execute be instruction

end else if (state == state_be1) begin
    // transfer addr1 to address
    addr1_drive = 1'b1;
    address_write = 1'b1;
    next_state = state_be2;

end else if (state == state_be2) begin
    // transfer mem[addr1] to alu_op1
    memory_drive = 1'b1;
    alu_op1_write = 1'b1;
    next_state = state_be3;

end else if (state == state_be3) begin
    // transfer addr2 to address
    addr2_drive = 1'b1;
    address_write = 1'b1;
    next_state = state_be4;

end else if (state == state_be4) begin
    // transfer mem[addr2] to alu_op2
    memory_drive = 1'b1;
    alu_op2_write = 1'b1;
    next_state = state_be5;

end else if (state == state_be5) begin
    // if (alu_op1 == alu_op2) take branch
    if (alu_equal_out == 1'b1) begin
        next_state = state_be6;
    end else begin
        next_state = state_fetch1;
    end
end

```

```

end else if (state == state_be6) begin
    // transfer addr0 to pc
    addr0_drive = 1'b1;
    pc_write = 1'b1;
    next_state = state_fetch1;

end

```

Assembly language

Recall program to compute difference between mem[20] and

mem[21]

mem[0] 15
mem[1] 12
mem[2] 20
mem[3] 21

mem[4] 2
mem[5] 22
mem[6] 20
mem[7] 21

mem[8] 13
mem[9] 16
mem[10] 0
mem[11] 0

mem[12] 2
mem[13] 22
mem[14] 21
mem[15] 20

mem[16] 0
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.
 $\text{mem}[20] = \text{mem}[20] + 1$?

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 addr0 addr1 addr2
```

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)

Comments marked by // (rest of line is ignored)

Blank lines ignored

Unspecified locations filled in with 0

What's the difference?

```
diff_less    blt diff_less diff_x diff_y
diff_end     sub diff_result diff_x diff_y
              be diff_end 0 0
diff_end     sub diff_result diff_y diff_x
diff_end     halt
```

How to initialize variables (diff_x, diff_y)?

Note the naming convention. Prefix all labels with name of file. Why?

If-then-else structure

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

```
                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

Loop structure

Structure of a while loop

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

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

Structure of a do-while loop

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

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

Find the maximum of mem[0] through mem[15]

Calling functions in assembly language

Example: overall 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?

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

AND

A	B	AND(A,B)
0	0	0
0	1	0
1	0	0
1	1	1

- $\text{AND}(0,X) =$
- $\text{AND}(1,X) =$

OR

A	B	OR(A,B)
0	0	0
0	1	1
1	0	1
1	1	1

- $\text{OR}(0,X) =$
- $\text{OR}(1,X) =$

NOT

A	NOT(A)
0	1
1	0

E100 bit-manipulation instructions

AND (opcode 6) (bitwise and)

- $\text{mem}[\text{addr0}] = (\text{mem}[\text{addr1}] \& \text{mem}[\text{addr2}])$
- e.g. what is $9 \& 10$?

OR (opcode 7) (bitwise or)

- $\text{mem}[\text{addr0}] = (\text{mem}[\text{addr1}] | \text{mem}[\text{addr2}])$
- e.g. what is $9 | 10$?

NOT (opcode 8) (bitwise negation)

- $\text{mem}[\text{addr0}] = \sim(\text{mem}[\text{addr1}])$ (addr2 is unused)
- e.g. what is ~ 9 ?

SL (opcode 9) (shift left)

- $\text{mem}[\text{addr0}] = \text{mem}[\text{addr1}] \ll \text{mem}[\text{addr2}]$
- e.g. what is $9 \ll 2$?

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

SR (opcode 10) (shift right)

- $\text{mem}[\text{addr0}] = \text{mem}[\text{addr1}] \gg \text{mem}[\text{addr2}]$
- e.g. what is $9 \gg 2$?

- 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 16-bit number. How to tell if 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 an 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: DPDT_SW, 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)

I/O instructions

E100 programs communicate with an I/O controller to generate output or read input

- in, out instructions
- program issues commands to the controller to cause them to do something. E.g. program tells graphics card to clear the screen
- Instruction specifies which I/O controller and which signals to communicate on by specifying the “port”

IN (opcode 18)

- `mem[addr1] = data from I/O port addr0`
- E.g. port 0 is used to get input from `DPDT_SW[15:0]`

```
        in 0 num
num     .data 0
```

- in this case, the I/O controller just feeds in the value from `DPDT_SW`

OUT (opcode 19)

- send mem[addr1] to I/O port addr0. I/O port “holds” this value until the next OUT instruction to that port.
- E.g. port 4 is used to send output to HEX7-HEX4

```
        out 4 num15
num15 .data 15
```

- 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?

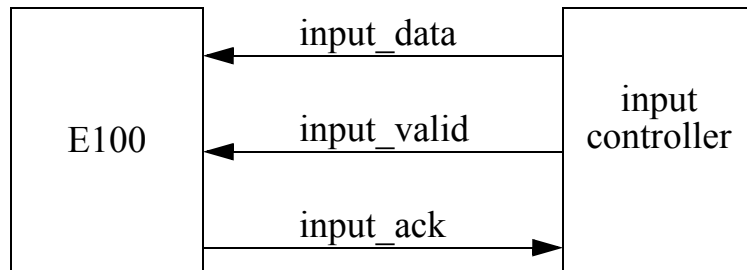
What problems did I encounter when trying to communicate numbers to you?

Communication protocols

Need a **protocol** to read numbers, even for something as simple and direct as DPDT_SW (port 0)

Signals for an input protocol

- `input_data`: the data that is being communicated from the I/O controller to the E100
- `input_valid`: I/O controller sets this when data is ready to be read by the E100
- `input_ack`: E100 sets this when it's seen the last input data

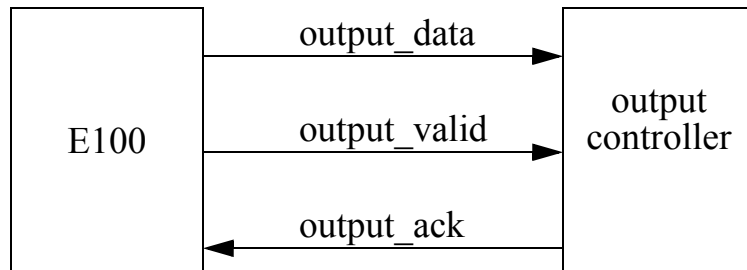


Protocol for receiving input from an input controller

- start with `valid==0, ack==0` (system is idle)

Signals for an output protocol

- output_data: the data that is being communicated from the E100 to the I/O controller
- output_valid: E100 sets this when data is ready to be read by the I/O controller
- output_ack: I/O controller sets this when it's seen the last output data



Protocol for sending output to an output controller

- start with valid==0, ack==0 (system is idle)

E100 simulator (ase100) simulates the DE2's I/O controllers

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?

Other protocols possible

- what protocol is used for conversation around a dinner table?