UIC – ECE 366: Computer Organization II - Fall 2018

Project 2: ISA Design

For this project, you will design your own *ISA (Instruction Set Architecture)* with its relevant software and hardware packages.

This will be a processor with 8-bit instructions (including 1 parity bit) and 16-bit data (registers and memory), which you will optimize for two programs (ME - "Modular Exponentiation", and BMC - "Best Match Count").

Your ISA design should be very clear with the following:

- a unique name
- which instructions to support, and how to encode each within 7 bits
- number of registers, general-purpose or specialized
- memory addressing modes: How are addresses of memory constructed / calculated? (This applies to both data and instruction memory accessing: load/store, and branch instructions).

General specifications:

- Instructions and data are stored in a separate memory hardware (both begin at address 0):
- Instruction memory: 8-bit byte addressable, PC is initialized (automatically) at 0. This memory should be as large as you need to hold one program. You can assume that each of the two programs will be separately loaded in, so each will start at location 0 of instruction memory.
- Data memory: 16-bit double-byte addressable with the following access limit: for any instruction, it can either load from or store to (but not both) the data memory exactly 16 bits. The data memory should be able to hold all the data needed for your two programs. Your program can change the values in the data memory.
- When implemented, this will be a single-cycle CPU, so you cannot assume to have a general "multiply" or "divide" instruction. If you want to have some special, "powerful" instructions, you need to show its implementation in ALU as combinational logic, not sequential ones.
- State any assumptions you make. You can assume that all the registers are initialized to be 0, but if any other numbers are needed in the registers, you need to put them there by some instruction.

Optimization goals:

1. High Speed in SW: (i.e., minimizing dynamic instruction count for one or both programs).
2. Low Cost in HW: (i.e., making your CPU's hardware design simplified).

[**Program 1: ME - "Modular Exponentiation"**]

Compute the result (R) for: $R = 6^P \% Q$.

Input: the 16-bit positive numbers P and Q are initially stored in data memory:

P = Mem[0]
Q = Mem[1]

Output: the 16-bit answer *R* should be written back into memory address 2 by your program:

Mem[2] = R

Levels of completion:

1) (80% grade) Assume a fixed Q = 17.
2) (100% grade) Support general 16-bit P and Q.

[**Program 2: BMC - "Best Match Count"t**]

Find out the Best Match score (S = [0, 16] where 16 indicates total match) and count (C: number of best matches) from an array (Pattern_Array[]) of one hundred 16-bit patterns, to a given target pattern (T).

Input:

The 16-bit pattern T is initially stored in data memory address 3:

T = Mem[3]

The array begins at data memory address 8:

Pattern_Array[1] = Mem[8]
Pattern_Array[2] = Mem[9]
Pattern_Array[3] = Mem[10]
…
Pattern_Array[100] = Mem[107]

Output: the 16-bit answers S and C should be written back into memory by your program:

Mem[4] = S
Mem[5] = C

Levels of completion:

1) (80% grade) Only need to support "total matching": S=0 and C=0 if no total matching is found.
2) (100% grade) Support in general any best-matching score

Submission components:

While preparing for your submission files, keep in mind it is your responsibility to make everything clear – as the person grading it has no knowledge about your ISA.

Include the following 6 files in your Bb submission:

- `project2_group_x_dis.py:` Python disassembler for your ISA
- `project2_group_x_p1_bin.txt:` machine code for P1 (EM)
- `project2_group_x_p2_bin.txt:` machine code for P2 (BMC)
- `project2_group_x_p1_asm.txt:` disassembler output for P1 (EM)
- `project2_group_x_p2_asm.txt:` disassembler output for P2 (BMC)
- **`project2_group_x_report.pdf`: a self-contain PDF report writeup**

PDF Report writeup components:

Part A) ISA intro

1. **Introduction**. This should include the name of the architecture, overall philosophy, specific goals strived for and achieved.
2. **Instruction list**. Give all the instructions, their formats, opcodes, and an example.
3. **Register design.** How many registers are supported? Is there anything special about the registers?
4. **Control flow (branches).** What types of branches are supported? How are the target addresses calculated? What is the maximum branch distance supported? Give examples of your assembly branch instructions and their corresponding machine code.
5. **Data memory addressing modes.** What addressing modes are supported for data memory? How are the addresses calculated? Give examples of your assembly load / store instructions and their corresponding machine code.

Part B) Answers to questions

1. Comparing to the sample of "My_straightforward_ISA", what are the unique features of your ISA? Explain why your ISA is better.
2. In what ways did you optimize for the two goals? If you optimized for anything additional, what and how?
3. What would you have done differently if you had 1 more bit for instructions? How about 1 fewer bit?
4. How did your team work together to accomplish this project? (Role of each team member, progress milestones, time spent individually and together?)
5. If you had a chance to restart this project afresh with 3 weeks' time, how would your team have done differently?

Part C) Software package:

1. **Algorithms (in assembly code) of the two programs.** Make sure your assembly format is either obvious or well described, and that the code is well commented.

   Extra credits: provide a convincing estimation:

   i. on the dynamic instruction count for P1 (ME) with P = ~1000 and Q = ~500
   ii. on the worst-case scenario of dynamic instruction count for P2(BMC).

2. **Machine code for each of the programs.** We will not correct/grade the machine code. You will also not be able to verify whether your code works correctly or not in this project (without a simulator). Therefore, you have to rely on the help of the disassembler, strive to make your algorithm simple and easy to understand, as well as pursue the sw-hw "codesign" – this will avoid putting tremendous complexity at either the software or the hardware end.

3. **Output of your Python disassembler for each program.** This should be a line-by-line explanation of the machine code, what is done by each line of code.

4. **Python code for your ISA's disassembler.**

Part D) Hardware implementation:

1. **ALU schematic.** A hierarchical sketch of your Arithmetic Logic Unit which implements whatever computation that your ISA instructions use (See textbook ch 5.2.4).

2. **CPU Datapath design.** A schematic including your register file, ALU, PC logic, and memory components (see textbook ch 7.3.1).

3. **Control logic design.** Decoder truth-table indicating how each control signal is generated from an instruction (see textbook ch 7.3.2).