CSE 141L Lab 1
9-Bit Instruction Set Graphics Architecture

Due Wednesday, January 23.

In this lab, you are the instruction set architect for a new special-purpose graphics processor. You will design the hardware for that processor in subsequent labs. This will be a 9/16-bit (9-bit instructions, 16-bit data types) processor which you will optimize for one application (described later). For this lab, you will design the instruction set and instruction formats, and code the program to run on your instruction set. Given the extreme limit on instruction bits, the target program and its needs should be considered carefully. The best design will likely come from an iterative process of designing an ISA, then coding the program, then redesigning the ISA.

Your instruction set architecture should feature fixed-length instructions 9 bits wide. Your instruction-set specification should describe:
- what operations it supports and what their opcodes are.
- how many instruction formats it supports and what they are (in detail -- how many bits for each field, and where they’re found in the instruction). Your instruction format description should be detailed enough that someone could write an assembler (a program that creates machine code from assembly code) for it. If values in particular bitfields have special meaning, please tell us what they are.
- number of registers, and how many; general-purpose or specialized.
- the size of main memory.
- addressing modes supported (this applies to both memory instructions and branch instructions). That is, how are addresses constructed/calculated.

In order to fit this all in 9 bits, you will have to think through your addressing modes carefully; feel free to be creative and invent new ones – for this project that is likely to be a good idea. You will actually have three memory spaces (and distinct memory parts) that you will need to be able to address – instruction memory, data memory, and graphics memory (described later). You should also assume that the latter two are word addressable, where a word in this architecture contains 16 bits.

When implemented, this will be a single-cycle machine (meaning that you execute exactly one instruction every cycle, no more, no less), so realize that there is a limit to what can be done in a single cycle. For example, assume you cannot read two memory locations (or read one and write one) from the same memory space in a single cycle. However, you will be able to design a register file that reads two and writes one register in a cycle (or more). You will be allowed to do a pretty much unlimited amount of combinational logic per cycle, though.

To simplify the ISA design process, you need only optimize for the following two goals:
1. Minimize dynamic instruction count (i.e., the number of instructions executed during the running of a particular program).
2. Simplify your processor hardware design.

You are welcome to also optimize for other things (e.g., cycle time, ease of pipelining), but if you do so, we will expect you to discuss that optimization intelligently, and these two goals should still take highest priority. Designs that do a good job of optimizing for goal 1 will receive higher credit, but you may still choose to emphasize goal 2 for your own benefit.

Generic ISAs (that is, ISAs that will execute other programs just as efficiently as those shown here) will be seriously frowned upon. We really want you to optimize for this program only – if you end up with an ISA that would be effective for other random programs, you haven’t done your job.

You will turn in a lab report no more than eight pages long (excluding the program listing). The report will answer the following questions. In describing your architecture, keep in mind that the person grading it has much less experience with your ISA than you do. It is your responsibility to make everything clear.
Your lab report will be structured as follows:

I. Introduction & general discussion.

II. Addressing modes supported – describe all addressing modes supported, for each type of memory, including how the addresses are actually calculated. Mention the address range each mode is capable of addressing. Give examples if necessary.

III. ISA description – give all details necessary to write an assembler. Describe each instruction format supported, and all opcodes supported. For each opcode, give the assembly code of a sample instruction.

IV. Example – give a sample instruction, and show how it would be assembled into a 9-bit machine instruction.

V. Control Flow – Describe all branches/jumps/procedure call instructions and how target addresses are computed, how conditional branches are handled, etc.

VI. Questions – Please give thoughtful and accurate answers to the following questions:

1. How many registers are supported? Is there anything special about the registers? Are they general purpose or specialized?

2. What is the maximum branch distance supported?

3. In what ways did you optimize for dynamic instruction count?

4. In what ways did you optimize for ease of design?

5. If you optimized for anything else, what and how? (It’s OK if you didn’t)

6. Your chief competitor just announced a load-store ISA with three operands, two registers (i.e., a 1-bit register specifier), and 64 instructions (6 opcode bits). Tell me why your ISA is better.

7. What do you think will be the bottleneck in your design? That is, what don’t you have that you will miss the most if you were to have to write other programs.

8 (a). What would you have done differently if you had 2 more bits for instructions?

(b). 2 fewer bits?

9. Can you classify your machine in any of the classical ways (e.g., stack machine, accumulator, register, load-store)? If so, which? If not, give me a name for your class of machine.

10. Write the program described below. Give assembly instructions. Make sure your assembly format is either very obvious or well described, and well commented. If you also want to include machine code, the effort will not be wasted, since you will need it later. We will not correct/grade the machine code. State any assumptions you make.

11. What is the dynamic instruction count of your program, assuming the following input:

   LINE, 3, 3,100,12, CIRCLE, 66, 72, 14, FILLEDBOX, 59, 12, 83, 21, END

The program. Your processor will be a special-purpose graphics processor, whose only function is to execute the following program as efficiently as possible. This program takes a description of graphic objects and “rasterizes” them, that is, sets individual pixels in a graphics buffer on and off to display a raster image of the graphic object. Your program will be able to interpret descriptions of lines, circles, and boxes, stored in memory. The graphic description will be a series of tokens and values, starting at data memory address 0. The allowable tokens are:

<table>
<thead>
<tr>
<th>Token</th>
<th>Meaning</th>
<th>Following words</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>END</td>
<td>None</td>
<td>This is the last token of the graphic specification. Program can halt.</td>
</tr>
<tr>
<td>0x1</td>
<td>LINE</td>
<td>X1, Y1, X2, Y2</td>
<td>Draw line from (X1, Y1) to (X2, Y2).</td>
</tr>
<tr>
<td>0x2</td>
<td>CIRCLE</td>
<td>X0, Y0, Radius</td>
<td>Draw a circle centered at (X0, Y0) with Radius Radius.</td>
</tr>
<tr>
<td>0x3</td>
<td>BOX</td>
<td>X1, Y1, X2, Y2</td>
<td>Draw a box with horizontal and vertical lines, with one corner at (X1, Y1) and the opposite corner at (X2, Y2).</td>
</tr>
<tr>
<td>0x4</td>
<td>FILLEDBOX</td>
<td>X1, Y1, X2, Y2</td>
<td>Draw a box, but with all pixels inside the box turned on (set to 1).</td>
</tr>
</tbody>
</table>

Your program will continue reading tokens and values and drawing into the graphics buffer until it reads the END token.

Thus, this sequence of values, starting at address 0, will draw a big X and a filled square in the center:
All values in the graphics description will be unsigned integers. The graphic description (“input”) will be no longer than 225 words. If you need to access main memory to store temporary values or for scratchpad space, you will need to use addresses greater than 225.

Error handling. Pixels outside the drawing area should not cause an error, but should just not be drawn. Illegal tokens should cause the program to halt. It’s fine to just assume that you’ll see an END token or an illegal token before you get to address 225. No other input errors should be possible.

Drawing area. The graphics memory will contain 16384 bits, describing a 128 by 128 (by 1 pixel deep) area. A bit set to 1 indicates the pixel is on (black), and 0 indicates the pixel is off. The correlation between words in the graphics memory and pixels in the drawing area is shown here:

So, for example, pixel (33, 25) is on if bit 14 of the word at address 202 of the graphics memory is set to 1.

Algorithms to draw lines and circles without floating point types and without multiplication and division are included in another handout. Ignore these algorithms at your own peril.

Some things to consider:
1. Make sure your ISA includes a HALT instruction. We’ll tell you what it actually does later, but it is the last instruction executed by your program.
2. Use subroutines when necessary to avoid unnecessary code duplication. It will also increase readability, and facilitate the division of labor within groups. I expect, at the least, subroutines for things like draw_pixel(x,y), and maybe even draw_line(…), draw_circle(…), etc. So you’ll at least need some kind of procedure call (but it should be simple and efficient!).
3. Be aware that you won’t really know exactly what branches you need to support (in particular, what branch targets/distances you need) until you are done coding and tuning the rest of the ISA.
4. Keep in mind that ease of programming, similarity to real architectures, and generality are not part of your design goals.
5. You will have parts of your program that have a much bigger impact on dynamic instruction count than others.