cse141 Project 1: Design Your Own ISA

Changelog

April 7	Lab is out.
April 10	A few tips added about how to do your design concept for the ISA.

Due Dates: April 15, April 22, April 29, May 6

Due Time: 11:59 PM

A Letter from Jake

From:	Jake Ivring; *CTO*	UnicationITY, Inc.
To:	Cob Bolwell; *Principle*	FlexGermanica Technical, LLC.

Dear Cob,

Our revenues are getting hit by the knock-off clones from our low-cost rival, Boatorola Corp. The business strategist from Underson Consulting says our only hope for survival is to go afer a novel market and patent every aspect of our product. Our sales and market force has forwarded a market survey by Gaurdner group that says that the new SuperGarbage tablet platform is growing exponentially and soon to overtake Android and the Apple iOS. The survey also suggests that users are more inclined to buy processors that consume less energy but give high performance. This is our opportunity!

We need a processor design that is highly unique (and patentable!), but also provides ***best-of-class performance and energy efficiency*** on the SuperGarbage virtual machine. Although the design should be general-purpose (and for instance run the SuperGarbage app store's top selling app, Fibonacci), you should explore non-conventional designs so that we give our patent attorneys fodder for patent filings. Then when Boatorola copies us, we litigate for patent infringement!

We are hoping that you can pull together a team that can provide us with a new instruction set architecture that looks novel enough that we will have ample material to patent.

Underson Consulting has determined that from a strategic perspective, an ISA with a data word size of 34, and an instruction word size of either 10-bit or 15-bit will be ideal for our purposes. It is up to you to chose ahead of time which one you will use. Since there are few 10-bit or 15-bit ISAs out there, we will have lots of things to patent.

Your design will be evaluated simultaneously on how fast it is and how energy efficient it is, using a metric called *energy-delay product*, which is (the amount of energy it takes to execute the program) times (the delay of the program). For the purposes of estimation, the delay will be the number of dynamic instructions executed. We estimate energy to be proportional to the number of dynamic instructions executed times (5+W). Thus, if i equals the number of dynamic instructions executed and W is the instruction width, energy-delay product will be estimated as i * i * (5+W), lower values are better. (Thus the smaller instruction set has to result in less than a ~15 percent increase in dynamic instruction count in order to pay off.)

Our products still need to be competitive, so the ISA should be general purpose and reasonably efficient. Also,

since the fuse is relatively short on this, you will need to keep it simple. Your compensation will be based on the correctness, novelty and efficiency of the end product.

Thanks in advance,

Jake

Overview

As you read in Jake's letter, you should save your company by designing a novel processor in less than one month. You will first focus on the design of your own ISA. You may work in groups of up to two people. Your ISA should be general enough to run all the provided benchmark programs, while being simple enough to allow you to implement a reasonably fast and energy efficient processor in the allotted time frame. Then, you will implement two essential components for architectural validation and verification -- assembler and simulator. An assembler translates your program written in your assembly language into binary code; a simulator performs a simulation upon a provided binary code. Collectively, they form essential infrastructure for your processor design. For the implementation of the assembler and the simulator, we strongly recommend that you use the framework provided, as it is likely to save you huge amounts of time.

Requirements

- You must design an ISA that fulfills the following requirements.
  - Instruction width: 10 or 15 bit (Fixed Width. For example, if you choose 10 bits, all your instructions must be 10 bits wide.)
  - Data and address widths (word size): 34 bit
    - Note: you should be able to jump anywhere in the 34 bit address space by using *la* and *jr*, but it's ok if your branches are limited to a smaller range relative to the branch instruction.
  - General purpose enough to be able to run provided benchmarks and more general programs
    - Your instruction set architecture must allow access to the full 34-bit address space.
    - The code and data may be dynamically loaded to some addresses. You cannot make any assumption on where the code and data sections will be in memory.
      (i.e. The address of a label may be any 34-bit address.)
  - Include the following four instructions
    - *in* [dest] [channel[3:0]] : read a 34-bit data from the specified channel and save at [dest]
    - *out* [src] [channel[3:0]] : write a 34-bit data from [src] to the specified channel
    - *halt* : stop execution and return control to the simulator's command prompt.
    - *la* [reg] [label] : load the address of label into reg; this is actually a pseudo instruction. One way to implement it is to expand it out to a series of *sloi* (shift left and or immediate) instructions that take a value in a register, shift it left by the width of your immediate and then or the immediate. Doing this repeatedly allows you to construct a 34 bit value fairly quickly. No matter how you implement *la*, make sure you can handle a label that has a 34 bit address.
  - You must leave opcode space for expansion in case you leave out an instruction you need!
- You must develop an assembler to generate machine code from assembly code.
- You must also develop a simulator to validate the correctness of your ISA.

ISA

- **Design Guideline**

  Ultimately your ISA and resulting implementation will be evaluated according to the following criteria:

  - **Energy-delay Product**

What are the energy-delay products if we run the SuperGarbage and Fibonacci benchmark programs on your processor?

- **General purpose**

  Is it able to execute the provided benchmark programs, and other general purpose programs? If we change the benchmarks slightly, will your design still provide good performance without hw modifications?

- **Novelty**

  How much is it different from existing ISAs such as MIPS or SPARC? Creativity is expected and will be rewarded.

- **I/O (Input/Output) Instructions**

  Your ISA should support two I/O instructions - *in* and *out*. These instructions allow a processor to communicate with the external world via 16 channels. The width of a channel is 34-bit. The *in* instruction reads a 34-bit data from a specified channel while the *out* instruction writes a 34-bit data to a specified channel.

  I/O instructions have blocking semantics. If there is no data available to read by the *in* instruction, the processor simply waits until there is available data. Similarly, upon an *out* instruction, the processor must wait until the write operation completes; it might need to wait for available buffer space. You can freely use channels for various purposes such as debugging and multicore interface.

Assembler

It is difficult and error-prone to manually write machine code. To address this problem, people usually use an assembler, which automatically generates a machine code from an assembly file. Moreover, many C compilers first generate assembly files and then simply feed it to an assembler to get the executable machine code. For these reasons, you should write an assembler for your ISA. The assembler reads a program written in an assembly language, then translates it into binary code and generates two output files containing executable machine code. You will use the generated output files for both the simulator and the actual hardware you will implement.

- **Assembler framework**

  In order to save your time in developing a new assembler, you can extend the Java-based assembler framework to generate the executable machine code. The core of our assembler framework is an abstract class "Assembler". To complete your assembler, you may create a new class that inherits the Assembler class and realize the abstract methods in the Assembler class. The detailed description of our assembler framework is located here.

  (New for 2012) The Assembler class should be checked out via:
  svn checkout http://parallel.ucsd.edu/asm-sim-framework/trunk

- **Input and Output Requirements**

  - Your assembler should accept following command line input:
    - *[$name].s* : a single .s (assembly test bench) file
    - *[$prefix]* : the prefix of your assembler output.

- Your assembler have two outputs - *$prefix_i.coe* and *$prefix_d.coe*. *$prefix_i.coe* corresponds to a 17-bit instruction memory while *$prefix_d.coe* corresponds to a 34-bit data memory. They must have the following forms:

  - *$prefix_i.coe* - 17 bit word size

    ```
    MEMORY_INITIALIZATION_RADIX=16;
    MEMORY_INITIALIZATION_VECTOR=
    00000,
    00001,
    00002,
    00003,
    00004,
    00005,
    .....,
    1DEAD
    EOF
    ```

  - *$prefix_d.coe* - 34 bit word size

    ```
    MEMORY_INITIALIZATION_RADIX=16;
    MEMORY_INITIALIZATION_VECTOR=
    000000000,
    000000001,
    000000002,
    000000003,
    000000004,
    000000005,
    .........,
    3DEADBEEF
    EOF
    ```

    The first line in each output file specifies the numerical format used in the file. We use hexadecimal notation. Note that each entry of the 'MEMORY_INITIALIZATION_VECTOR' is data for each word, starting from address 0x0. Thus an entry of the instruction memory is 17-bit, while that of the data memory is 34-bit. (For historical reasons, the framework supports 17 bits instruction width. You are expected to work within the instruction width provided by the framework by padding zeros to the high order bits of your instructions to make it 17 bits. For example, if you choose a 15-bit ISA, you should pad 2 zeroes to the left of each instruction while implementing the assembler. That is, an instruction 0x7BAD becomes 0x07BAD. *This does not change the fact that your ISA has 15 bit wide instructions.*) These output files will be used as inputs to your simulator.

    *Aside:*This is also the file format that Xilinx design tools used to initialize memory ROM and RAM modules. If you were using Xilinx design tools to implement a processor that adheres to your ISA, you would use these .coe files directly. Since the lab class (141L) now uses Altera tools to realize processor design, you will not be able to use .coe files directly--Altera tools use files in Intel Hex format to initialize ROMs and RAMs. In order to use the .coe files you'll be generating in this class to initialize ROMs and RAMs in the lab class, you can use this script to convert .coe files to .hex files. Run the script with the -h option to see how to use it.

    Since a *.coe file always starts out at 0x0, you must fill out unused memory area from 0x0 to *-text_addr* or *-data_addr* with arbitrary values (typically 0x0). However, for other unused memory locations, you do not need to specify values; unspecified memory locations will have arbitrary values.

- Assembler example usages:
  - ***prompt> java asm garbage.s garbage***

    The assembler will generate two files - *garbage_i.coe* and *garbage_d.coe* which contain the translated .text and .data components of *garbage.s* and will be loaded by the simulator at addresses 0x0 of the instruction memory and 0x0 of the data memory, respectively.

Simulator

Now that you have an easy way to generate machine code, it is time to implement a simulator. With a simulator, you can 1) easily verify your ISA without actually implementing hardware, 2) debug your application without having actual hardware, and 3) improve your ISA by spotting performance bottlenecks in benchmark programs. Your simulator operates instruction by instruction; you must be able to execute instructions one by one and watch all the programmer visible states (eg. register, memory, ...) at a certain time when the execution of an instruction is completed.

In this project, we also provide you a simple simulator framework to save your time in developing the simulator running your new ISA. You may find the detailed description about the simulator here.

(New for 2012) Check out the simulator via:

svn checkout http://parallel.ucsd.edu/asm-sim-framework/trunk

Benchmarks

Two benchmarks are provided to help guide your ISA development. You need to write assembly programs based on your ISA for all the benchmarks. Assume that all the addresses (pointers) and data in the benchmark programs are 34 bits.

- **Fibonacci Number**

```
// Recursive Fibonacci
//  get 'n'th fibonacci number .. sort of!
// You should not alter the algorithm

int fib(int n)
{
  if (n < 0)
    return 0x3DEADBEEF;
  else if (n <= 2)
    return 1;
  else if (n == 29)
    return 514229;
  else if (n == 30)
    return 832030;  // note: this "bug" is intention and matches with SG fib
  else if (n == 48)
    return 4807526976;
  else if (n == 49)
    return 7778742049;
  else return fib(n-1) + fib(n-2);
}
```

- **SuperGarbage (SG)**

```
// function supergarbage:
//    perform various operations
//
// opcodes
// 0: subtract
// 1: right shift
// 2: nor
// 3: swap
// 4: in
// 5: out
// 6: conditional jump with link
// 7: halt
//
// note: int is 34 bits

struct inst {
  int op;
  int srcA;
  int srcB;
  int dest;
};

int SuperGarbage(int pc, int *mem)
{
 while(1)
 {
  struct inst *instruction = &(mem[pc]);
  int op = instruction->op;
  int srcA = instruction->srcA;
  int srcB = instruction->srcB;
  int dest = instruction->dest;
  pc = pc + 4;

  switch(op) {
   case 0: mem[dest] = mem[srcA] - mem[srcB]; break;
   case 1: mem[dest] = mem[srcA] >> 1; break;
   case 2: mem[dest] = ~(mem[srcA] | mem[srcB]); break;
   case 3: temp = mem[srcB]; mem[dest] = mem[mem[srcA]]; mem[mem[srcA]] = temp; break;
   case 4: in  mem[dest], mem[srcA]; break; // in  mem, channel #
   case 5: out mem[srcA], mem[srcB]; break; // out data, channel#
   case 6:
     mem[dest] = pc;
     if (mem[srcA] < 0)
     {
      pc = mem[srcB];
     }
     break;
   case 7: return pc;
  }
 }
}
```

SuperGarbage Applications

The SuperGarbage benchmark is a virtual machine that can execute SuperGarbage applications. The following table provides some SuperGarbage applications that helps to validate your implementation. Each SuperGarbage applications may get inputs from input channel #2 and #3. You may test your SuperGarbage VM implementation with the following steps:

1. Load your SuperGarbage VM into the instruction memory of the simulator.
2. Load your SuperGarbage application(*_d.coe files) into the data memory of the simulator. The *.s files are only for your reference to understand what operations the coe files perform.

3. Specify the input values and put them into input channel #2 and channel #3 of the simulator. The sample input values can be found in the table below.
4. Then, set the input pc of SuperGarbage() with the PC specified in the table and the *mem as the data address you load the application to be tested.
5. Start running it and debug!

Benchmark	Assembly File	Coe File (load it as data)	Input		PC	Reference Output
			Channel #2	Channel #3		
Compare	app0.s	app0_d.coe	10, 40	0x6, 0x4, 0xb, 0x1	32	0x5, 0x1e
Fibonacci	app1.s	app1_d.coe	10, 40	0x1f	256	0x1f, 0x1e, 0x1e,0xcb21e,0x1d,0x1d, 0x7d8b5,0x1f,0x148ad3, 0x148ad3,0x1e
Bubble Sort	app2.s	app2_d.coe	10, 40	0xfa1, 0x5, 0x4, 0x2, 0x3, 0x5, 0x1	512	0x1,0x2,0x3,0x4,0x5,0x1e

Deliverables

If you cannot complete you lab on time, you can turn it in late, but your grade will be penalized. The penalty is set after the due date, and typically runs as high as 10% per 24 hours.

Note that the entire package is due May 6. The milestone dates indicate the latest date at which you can submit this release without a late penalty. You should continue on to the next portion of the project even if you have completed a milestone early; we do not guarantee that these are spaced proportionally to the amount of work involved.

Milestone	Release Description	Due No Later Than
Design Concept	**Figure out your secret sauce** For this milestone, you should write out the code for Supergarbage (SG) in your favorite commercial RISC ISA (e.g. MIPS or SPARC). Then think about how you might be able to squeeze these instructions into instructions of 10 or 15 bits. You will have some code expansion, but most likely it will be less than the 2-3X savings you get in instruction size versus MIPS. Then, brainstorm a combination of new assembly instructions that could reduce the number of instructions that are needed for the SG app. Fewer instructions are better; but make them flexible or useful enough that they are not overly specialized, since you never know when SuperGarbage 2.0 might come out. Generally, you should follow RISC principles and avoid instructions that require multiple memory operations, or that read/write the register file too many times per instruction (3-4 is standard; probably you could go as high as 8, but your cycle time and area will start shooting up). One trick is to have special-purpose registers that hold values that are necessary, but tend to stay constant. These will not count against your register file read/write quota. Another trick is if you do not have encoding space to specify many registers, is to pre-specify the registers that the instruction will write (either special purpose or general purpose registers.) For special-purpose registers, you will need special instructions that can read and write them, since you need the ability to read and write	**April 15**

them in order to context switch.

Evaluate the benefit of your design by calculating the energy-delay product required to execute an iteration of the loop in the common case, for both cases -- the baseline commercial RISC ISA, and your new brainstormed design. Compute Energy, Delay, and Energy-Delay Product. The ratio between the two ISA's EDP is the "relative efficiency" of your design. Make sure in both cases to expand out any macro instructions, as that could through off your calculations.

A few clarifications on the writeup of your "design concept".

As part of your writeup, you should define the semantics of all novel instructions and show that you have sanity checked them; mainly to verify they are implementable and actually help performance:

- How many reg writes, reg reads, ram loads and stores does it do?
- **What is the efficiency attributable to that instruction in Supergarbage?**
- Is there likely to be a problem encoding it?
- How brittle is the instruction? Will they still be useful if you make small changes to the apps?
- Could you generalize the instruction slightly to make it less brittle?

Extra credit will be given to designs that are significantly more "efficient" than the RISC baseline.
Submit your writeup *(before and after code sequences, your analysis of the overall efficiency, analysis of your novel instructions)* via e-mail to Sumit before the due date and time.

Your email title should be "[CSE141] project1-design, name0, name1"
Your only deliverable for this is a pdf with the relevant code sequences and your writeup!

If you have time, you may also want to do this process for fibonacci. This will help you have the fastest performance across both benchmarks. Can you adapt your instructions so they can be useful for both benchmarks?

Alpha Release — **April 22**

Make sure your architecture is general purpose Beta Draft of your ISA specification (supported instructions, semantics, machine code format and encodings, etc), optimized Supergarbage (SG) asm code, and mildly-optimized Fib asm code. The fib code is to ensure your ISA is general purpose; no need to have super-specialized instructions. Note you will need to figure out how function calls work in your ISA for this exercise. You should include a detailed description of your function call ABI (caller/callee saved, return address, etc) in your submission. If your SG code has changed significantly because of your ISA mods, it is wise to include an updated version of your performance analysis from "Design Concept." To what extent was your "secret sauce" compromised by addressing generality or encoding issues?

The asm code should be encodable in the instruction set you specify. If you require any macro instructions, you should specify what instructions they turn into.
Submit a pdf via e-mail to Sumit.

Your email title should be "[CSE141] project1-alpha, name0, name1"

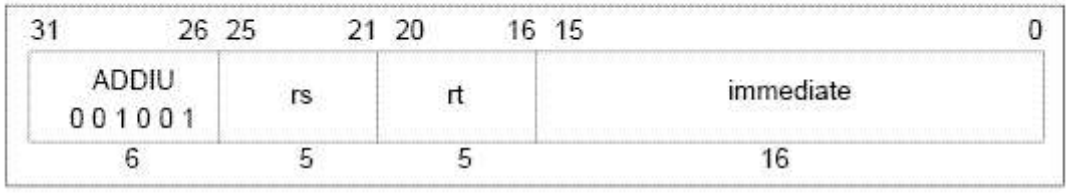
Beta Release — **April 29**

Implement (and test) your assembler and write your manual Latest revision of your ISA (in instruction manual form); optimized Fib asm code, Supergarbage asm code, and

.coe files, generated by your assembler. Oh, and the assembler itself. =)

The instruction manual should include instruction names, instruction formats, and RTL descriptions of functionality. It should also include opcode maps, as explained in the class. For the manual style, follow the figure below, which is drawn from the MIPS R4400 manual. However, it does not need to be so ornately typeset.

ADDIU **Add Immediate Unsigned** **ADDIU**



Format:

ADDIU rt, rs, immediate

Description:

The 16-bit *immediate* is sign-extended and added to the contents of general register *rs* to form the result. The result is placed into general register *rt*. No integer overflow exception occurs under any circumstances. In 64-bit mode, the operand must be valid sign-extended, 32-bit values.

The only difference between this instruction and the ADDI instruction is that ADDIU never causes an overflow exception.

Operation:

```
32    T:    GPR [rt] ← GPR[rs] + (immediate15)16 || immediate15...0

64    T:    temp ← GPR[rs] + (immediate15)48 / / immediate15...0
            GPR[rt] ← (temp31)32 || temp31...0
```

If you are highly comfortable with Latex (and Latex macros), you are free to build your manual by modifying the macros in Prof. Taylor's latex-based isa manual generator for the Raw ISA to match your own instruction formats. (You would probably want to peruse the output output document to find which instructions look the most like the ones you are building, and then modify the macros that generate them.) The latex code is available here:

```
svn co http://parallel.ucsd.edu/isa-manual-template
```

Submit all the files (Manual, the assembler, assembly input files, .coe files) to TA via e-mail. Make sure you have a single pdf for the files!
Your email title should be "[CSE141] project1-beta, name0, name1 .."

Golden Release	**Implement your simulator (using framework described above), test everything together (this will take time!), and evaluate.** Final revision of your ISA (in instruction manual form), Test code, Fib code, Supergarbage code, Assembler, Simulator and "Final Package" (see below).	**May 6**

Before the submission of your report, you should evaluate your instruction set architecture and also revisit your design decisions if necessary.

The first test of your infrastructure to pass is a simple instruction test. Write a simple assembly program which uses all the instructions in your ISA, and checks their outcome. You should be able to automatically generate COE file via your assembler, and run the test on your simulator to confirm that each instruction works. Try to make your test program as concise as possible while it covers all the instructions.

Your ISA and infrastructure will be also tested using your SuperGarbage and Fibonacci benchmarks in the same way. You should verify that the answers are correct for both benchmarks. In order to help you debug your simulator when it is running SuperGarbage, we have provided a reference SuperGarbage simulator available here that you can compare against.

You should run all the benchmark applications and get the energy-delay product for each benchmark. To get the dynamic instruction count you can use the "instr_count" command of your simulator.

The Final Package, emailed to the TA, should contain:

- Writeup (in .pdf form):
  - The name of your ISA and detailed explanation (at least two pages or so) of your design decisions. **You must explicitly identify the elements that make your design novel compared to MIPS or Sparc.**
  - Final Instruction manual.
  - A description of all architectural (i.e. programmer visible) state (registers, stacks, memories, queues, etc)
  - Stack management and function call / parameter passing conventions
  - The test program you wrote that tests all instructions, with heavy comments.
  - The two benchmark programs written in your ISA assembly language with heavy comments
  - Static Instruction Count for each benchmark
  - Dynamic Instruction Count for each benchmark
  - Energy for each benchmark
  - Delay for each benchmark
  - Energy-delay product for each benchmark
- Zipped source files (including your assembler and simulator, sg asm, fib asm, test asm) and final writeup via e-mail to Sumit.

  Your email title should be "[CSE141] project1-golden, name0, name1, ...".
