Design Process Journal

Milestone 1:

➔ Meeting 1:

In our first meeting, we started with a discussion on registers that we'll use. We based our decisions on our own experiences; we mainly use 2 arguments being passed to functions, so in making the common case fast and in light of our 16-bit constraint, 2 argument registers were determined. Additionally, we found a hard-wired-to-zero register very useful in branches, jumps, arithmetic, and a host of other commands, so that was a logical addition. Discussing the rest of the registers, we decided that using 16 registers would provide for the most function. The rest of the registers would be: 6 temporary registers, 3 save registers, 1 to store the stack pointer, 1 to store the return address, 1 to store the return value, and 1 assembly temporary. This would allow for the greatest amount of register flexibility, and the assembly temporary will allow for pseudo-instructions, which would greatly improve our use.

➔ Meeting 2:

In our second meeting, we started writing out all the different instructions that we knew were going to be vital to our processor. We put them into 3 categories: arithmetic, memory, and control

- add, sub, and. or, addi, andi, ori, slt ---- all op code starts with a 0
- lw, sw, lui ----- all op code starts with 10
- beq, bne, j, jr ----- all op code starts with a 11

This allows us to check the first couple bits to know what kind of instruction we are trying to accomplish.

We also started detailing the different types of instructions and how the 16-bits would be utilized in each different type. We created 4 types, an A-type, a Mem-type, a C-type, a J-type for logic.

- A-type - 4 bits op code, 4 bits destination register, 4 bits source register one, 4 bits source register two. This takes care of all the arithmetic instructions.
- Mem-type - 4 bits op code, 4 bits destination register, 4 bits source register one, 4 bits immediate value. This takes care of all memory type instructions.
- C-type - 4 bits op code, 4 bits destination register, 8 bits of an immediate value. This takes care of the control instructions.
- J-type - 4 bits op code, 12 bits of an immediate value. This takes care of jump.

We also came up with a naming structure for our different registers for usability for the programmers. It is as follows:

- $0 - hard-coded to the value 0.
- $a0 - first argument register
- $a1 - second argument register
- $t0-$t3 - Temporary registers
- $s0-$s2 - Saved Registers
- $sp - Stack Pointer Register
- $ra - Return Address Register
- $rv - Return Value Register

- $at - Assembler Temporary
- $ce - Cause Register
- $se - Status Register

Register Assignments
Instruction Types

Milestone 2:

➔ Meeting 1:

In our first meeting, we fixed assembly code in the Euclid's algorithm program (mostly updated comments and addresses to branch/jump to, and changed numeric registers to named registers for ease of maintenance).
Now, if we want to change what register is given which name, we only have to change it in our name/number reference, rather than in all of our code.
We moved the jal instruction to an op code closer to the other jumps, and shifted the branching instructions to compensate.
We also wrote an assembler, which takes assembly code from a file and prints its machine code to the console (doesn't write to a file, since the only machine code we should need to see is just going in our documents; i.e. we can copy+paste it from the console). We no longer have to write machine code by hand.
RTL was created for each instruction, as seen here.
The data path components' inputs, outputs, sizes, and descriptions of tests were completed, along with a control wire design, though no formalized state machine for control has been created yet.
Multicycle Design
Instruction Format

➔ Meeting 2:

In our second meeting, the hardware components used for each instruction were identified. A description of each component was given and their core functionality in the big picture of our design. Along with the description, we also added what each component's input and output should be. Furthermore, each output includes a description of what component receives it as an input. Lastly, we included a simple test case for each component followed by detailed description of each.

Milestone 3:

➔ Meeting 1:

In this first meeting, we analyzed what components will need to be made for our datapath. We identified several parts: a 3-input 16-bit mux, 4-input 16-bit mux, 4-to-16-bit sign extension, 8-to-16-bit sign extension, 12-to-16 bit sign extension, 8-to-16-bit load upper immediate, and a 16, 16-bit general register file. We each began to implement these components, and then wrote unit tests that tested each component. The next step, is to fill a project with all of the necessary components, and begin to construct the datapath and test the addition of components. Our plan for this integration testing is as follows:

1. Begin with first RTL block, and test dataflow with common values.
2. Implement next step in RTL, and test dataflow with common values.
3. Repeat until the RTL diagram is fully implemented into the datapath.

Milestone 4:

➔ Meeting 0:

Discussed planning for handling interrupt driven input/output. We have several ideas:

- Memory-Mapped
- Syscall command that will handle control for reading input
- Input/Output read/write to separate register file that, using syscall or other command, those register values can be moved between the I/O reg file and the general reg file.

➔ Meeting 1:

In our first meeting, we began integration testing and control implementation. We've modified the integration testing plan to incrementally test with the addition of each component into the datapath, but construct the datapath by first implementing the first cycle in the RTL, and then the rest. For this meeting, the PC register and the PC Selection mux have been put into the path and tested. Control planning has begun, and will be a focus of the next meeting.

➔ Meeting 2:

Control is mostly implemented in verilog. This meeting was primarily with debugging the control and continuing integration testing. We've begun to add in the memory file and test its integration with the rest of the datapath (PC Register and PC OutputSelection Mux).

➔ Meeting 3:

We've continued to debug the control and have been updating our design documentation.

Milestone 5:

➔ Meeting 0:

We implemented our designs for interrupts. We ran into many stumbling blocks but now believe our way of handling interrupts to be successful.

➔ Meeting 1:

We have begun testing each and every instruction on our datapath. We started with the easy instructions. We have successfully tested add, addi, bne, beq, jr, jal, j, ori, and lui. Debugging these were rather time consuming but in the end we believe they work.

➔ Meeting 2:

Control is mostly implemented in verilog. This meeting was primarily with debugging the control and continuing integration testing. We've begun to add in the memory file and test its integration with the rest of the datapath (PC Register and PC OutputSelection Mux).

➔ Meeting 3:

We've continued to debug the control and have been updating our design documentation.

Milestone 6:

➔ Meeting 0:

We continued to perform integration testing and unit testing the designs for interrupt handling.

➔ Meeting 1:

We began testing the completed datapath, by testing each individual instruction in short code blocks. As bugs arose, we addressed them.

➔ Meeting 2:

We continued to test the complete datapath, finishing the tests of each individual instruction. We began to test relPrimes and euclid's.

➔ Meeting 3:

We concluded testing euclid's and relPrimes, moving on to simulating interrupts. More bugs were encountered and we addressed them.

➔ Meeting 4:

We continued and concluded testing interrupt handling, and began to collect data.

➔ Meeting 5:

We continued to collect data and began to port the program onto the Spartan board. We made the presentation and began to make the report.

➔ Meeting 6:

We concluded data collection and finished the report.