Milestone 1

October 1st, 2016

AJ Granowski, Ben Holtzman, Matt Moon, Tucker Osman

We've decided on an accumulator processor architecture to guarantee basic functionality before we add additional features and pizzaz. This will also simplify how op-codes are constructed and made to fit in a 16-bit space.

Estimated time distribution for this week:

- Decide on the registers and how they're accessed (1 day, everybody)
- Create the procedure call convention (1 day, everybody)
- Defining instructions and what they do (1 day, everybody)
- Create addressing modes (1 day, everybody)
- Define how instructions translate into machine code (1 day, everybody)
- Create and document programs and common fragments that use the architecture (1 day, everybody)

Actual time distribution for this week:

- Decide on the registers and how they're accessed (1 day, everyday)
- Create the procedure call convention (2 days, everybody)
- Defining instructions and what they do (2 days, everybody)
- Create addressing modes (1 day, everybody)
- Define how instructions translate into machine code (1 day, everybody)
- Create and document programs and common fragments that use the architecture (1 day, everybody)
- Create and test an assembler (3 days, AJ)
- Create and test a function call simulator (2 days, Tucker)

Design Decisions:

There were several big decisions made while designing the ISA and the register set. First off, we had to add a backup register to reduce calls to the stack for really simple, menial things. We also decided to pass the first argument in the accumulator to reduce the time needed to run a one-parameter function.

The origin of the U-Type comes from its original name of "unassigned type." But then we started using the unassigned portion and were too lazy to change the name. We also liked calling that field the "unction" field to differentiate it from your run-of-the-mill F-Type.

Also, because our immediates for I-Types are kind of small, we wanted the ability to access more than just 2^{12} words of memory, so we added a memory page that's stored in the upper 8 bits of the flag register. The middle 6-bits were reserved for interrupts and exceptions when we implement them in the design.

At the start, we did not have a `SKIPIF` instruction, opting to only have a `SKIPNIF`. It turned out that a `SKIPIF` instruction would be beneficial because to make the code readable and, in some cases, more efficient.

Finally, we decided to make an rudimentary assembler and function tester early on in order to make sure that code could be implemented using our supported operations. It became very useful when debugging the GCD code, as we didn't have to step through the program with a whiteboard step by step.

Milestone 2

October 12th, 2016

AJ Granowski, Ben Holtzman, Matt Moon, Tucker Osman

In this episode of Designing a Processor, we converted all of our instructions into steps at the register transfer level. We then broke it apart into different hardware components and described everything going in, out, and how it'll be tested.

Estimated time distribution for this week:

- Write the RTL for each instruction (2 days, everybody)
- Split the RTL up into reusable modules (1 day, everybody)
- Define the inputs and outputs for each module (1 day, everybody)
- Describe the component and how it's tested (1 day, everybody)

Actual time distribution for this week:

- Write the RTL for each instruction (1 day, everybody)
- Split the RTL up into reusable modules (1 day, everybody)
- Define the inputs and outputs for each module (1 day, everybody)
- Describe the component and how it's tested (1 day, everybody)

Design Decisions

We decided that this processor should be a single cycle for pure simplicity. While we were writing the RTL for the IO instructions, we realized we didn't have any way to specify the destination port for outport. We were against using the backup register, so we just decided to put the data on the stack and the destination port in the working register. This may be a tad bit slow, but that's okay.

We also changed the page flag to be 4 bits instead of 8 bits because 4+12=16, which is the size of our bus. We just wanted to keep everything 16 bits, including our addressable memory space. We also decided that all of the code belongs on the first page.

In a sudden flash of intelligence, we added a third bit to the comparison flags because there are 6 possible states that could be expressed by the skip instructions. Even though only one flag will be set at a time, it could be compared against one or two set flags (never three because that would be true if it was greater than, equal too, and less than.)

Milestone 3

October 19th, 2016

AJ Granowski, Ben Holtzman, Matt Moon, Tucker Osman

In a dramatic fit of misunderstanding, we did far more work than we needed. So we implemented most, if not all, of the components and their tests. We also made the datapath, which wasn't as hard as it could have been.

Estimated time distribution for this week

- Create the datapath (3 days, Tucker & AJ)
- Create the control signals (2 days, Tucker & AJ)
- Describe and implement some components (2 days, Ben)
- Describe and write tests (1 day, Ben & AJ)
- Verify the control signals work with the datapath (1 day, Matt)
- Write the English text required (1 day, Matt)

Actual time distribution for this week

- Create the datapath (1.5 days, Tucker & AJ)
- Create the control signals (2 days, Tucker)
- Describe and implement some components (2 days, Ben)
- Describe and write tests (1 day, Ben & AJ)
- Verify the control signals work with the datapath (1 day, Matt)
- Write the English text required (1 day, Matt)

The first step was to implement the datapath. At the outset, it looked quite daunting. But as we started laying wires down, it became increasingly easy because it seemed like instructions were just variants of each other. Control would handle all of the differences. But, in general, the datapath kinda built itself.

After we built the datapath, we went on to list the control signals to make the processor do what it needs to do. The control unit is basically going to be one massive switch statement. Verilog makes things so much easier than schematics, and Ben knows it very well. Making the control signals wasn't hard, but very very time consuming indeed.

The biggest change we made was to the ALU. The ALU decides if we skip or not, not the control unit. So if the skipnif, skipif, skipclr, or skipset conditions are met, then it'll make the PC go ahead an extra 2 to make the skip happen. Other than that, our general design from milestone 2 remained the same.

Milestone 4

October 19th, 2016

AJ Granowski, Ben Holtzman, Matt Moon, Tucker Osman

As a result of the plot twist in the previous milestone, we had to do very little work for this one. We already had most components written and tested, so all we had to do was finish up memory and get started on the integration plan. We also cleaned up the RTL we wrote in previous milestones to make it easier to read.

Estimated time distribution for this week

- Create and test memory (1 day, Tucker, Matt)
- Clean up design document (1 day, AJ)
- Clean up ALU test (1 day, AJ)
- Implement PC subsystem (1 day, Tucker)
- Write register file test (1 day, Ben)

Actual time distribution for this week

- Create and test memory (1 day, Tucker, Matt)
- Clean up design document (1 day, AJ)
- Clean up ALU test (1 day, AJ)
- Implement PC subsystem (1 day, Tucker)
- Write register file test (1 day, Ben)

Since we did so much extra work in the previous milestone, all we had to do for this one was to create and test a memory unit, test the register file, and get started on our integration plan. Since we had already done the memory lab, it didn't take long to get the memory unit up and running. However, due to memory limitations on the FPGAs, we could only have 10 bit memory addresses.

After getting berated multiple times for our RTL being repetitive and inefficiently written, we decided it was time to make a change. We now have a much neater diagram that clearly displays the RTL in such a way that very little is unnecessarily repeated.

No design decisions were made in this milestone. We are sticking to the design that we created in the first couple milestones, and we haven't found any reason to change it yet.

Milestone 5

November 2nd, 2016

AJ Granowski, Ben Holtzman, Matt Moon, Tucker Osman

During this installation of a milestone, we successfully assembled our datapath and began implementing our system tests. We have successfully executed several instructions on the datapath.

Estimated time distribution for this week

- Test ALU Subsystem (1 day, Matt)
- Tests datapath (2 day, AJ)
- Control unit test (1 day, Ben)
- Test PC counter fragment (1 day, Tucker)

Actual time distribution for this week

- Test ALU Subsystem (1 day, Matt)
- Tests datapath (1 day, AJ; 1 day everyone)
- Control unit test (1 day, Ben)
- Test PC counter fragment (1 day, Tucker)

We nearly completed our integration plan and got a working datapath. The ALU subsystem was assembled and tested, along with the control unit subsystem and PC counter.

In order to test the datapath, extra debugging inputs and outputs were added to the datatpath along with muxes to allow for input injection and output extraction. We found that we ran into infinite loops initially, but this was solved by clocking the register file.

Some design decisions we made were clocking the register file and the alu. We clocked the alu because the register updates on the falling clock edge, and the alu needs to retain its output until then.

Milestone 6

November 9th, 2016

AJ Granowski, Ben Holtzman, Matt Moon, Tucker Osman

In this milestone, we finished verifying every instruction and we successfully ran relPrime. We also took performance data of our processor.

Estimated time distribution for this week

- Verify instructions (4 days, AJ, Ben, Matt, Tucker)
- Collect performance data (1 day, Tucker)
- Set up I/O (1 day, Ben)

Actual time distribution for this week

- Verify instructions (4 days, AJ, Ben, Matt, Tucker)
- Collect performance data (1 day, Tucker)
- Set up I/O (1 day, Ben)

We now have a working datapath and control. We carefully wrote a test that used every instruction and verified their functionality along the way. We spend a lot of time staring at waveforms.

The only change to the datapath that we made was to put in an adder and a mux that incremented sp when we needed it for the popra instruction. This was necessary because sp gets decremented in that instruction, but we needed to use the original sp to retrieve a value from memory.

We also had to put a twice as frequent clock in front of the memory unit because it needs to read/write values in the middle of the instruction, so it needed a rising edge there.