Work log = Matthew Lyons

Summary:

September 28th, 2018 -- 90 minutes

Met with team, started instruction set

September 29th, 2018 -- 90 minutes

Met with team, continued on instruction set

September 30th, 2018 (morning) -- 30 minutes

Worked individually on assembly code fragments

September 30th, 2018 (afternoon) -- 105 minutes

Met with team, discussed major problem facing instruction set

October 1st, 2018 (morning) -- 15 minutes

Met with team and Sid, discussed various things

October 1st, 2018 (evening) -- 135 minutes

Decided on two accumulators with swap, worked on deliverables

October 1st, 2018 (night) -- 60 minutes

Finished documentation

October 3rd, 2018 (afternoon) -- 75 minutes

Worked individually; fixed formatting on documentation, added title page and table of contents, added quick reference table for instructions

October 5th, 2018 -- 45 minutes

Split up the RTL commands, decided against pipelining

October 6th, 2018 (morning) -- 30 minutes

Made the RTL for the commands I was assigned

October 6th, 2018 (evening) -- 90 minutes

Shared our RTLs with eachother, decided on some conventions, started the state diagram, shopping list, and datapath

October 7th, 2018 -- 60 minutes

Decided to turn the flag bit into an @ sign for readability. Made a big state diagram.

October 9th, 2018 -- 60 minutes

Wrote tests for RTL, finished up milestone

October 10th, 2018 -- 30 minutes

Meeting with Sid to discuss Milestone 2

October 13th, 2018 -- 85 minutes

Remade the ALU in Verilog for practice

October 15th, 2018 (afternoon) -- 45 minutes

Worked on implementing main memory in Verilog

October 15th, 2018 (evening) -- 45 minutes

Added support for setGreaterThan and setEqualTo to the Verilog ALU

October 16th, 2018 (afternoon) -- 45 minutes

Finished implementing main memory in Verilog, complete with tests

October 17th, 2018 (afternoon) -- 95 minutes

All components complete, datapath complete

October 19th, 2018 (afternoon) -- 30 minutes

Meeting with Sid to discuss Milestone 3

October 20th, 2018 (evening) -- 90 minutes

Fixed the stuff that went wrong in Milestone 3

October 21st, 2018 (morning) -- 60 minutes

Wrote up a finite state machine for the control unit (by hand)

October 22nd, 2018 (evening) -- 90 minutes

Worked on integration and the control unit, made plan to handle interrupts

October 23rd, 2018 (evening) -- 90 minutes

Worked on integration and the control unit

October 24th, 2018 (morning) -- 25 minutes

Typed up the handwritten finite state machine on the design document

September 28th, 2018 -- 90 minutes

No individual work was performed. All work was done as a group. However, I will still give my thoughts on all major discussion points.

We've decided on accumulator architecture primarily because Joy really wants to do it. Personally, I'm content as long as it isn't mem2mem, which I have a mostly-irrational bias against. I think stack would have been fun, and I think load-store would have been good, but accumulator hits a satisfying middle ground that I think will turn out well.

We're finding rather quickly that we need to use the stack for a lot of operations. Preferring not to access memory more than we absolutely have to, we're leaning towards having a mini-stack of registers that we use before dipping into the memory stack-- a sort of buffer stack. This would allow us to make use of a quick stack without the slowness of interacting with memory. Because of this haphazard fusion of accumulator and stack, we later decide to name our processor "Frankie," after Frankenstein's Monster.

We've begun fleshing out our instruction types. At first it was looking like we might get away with a 4 bit op code, but on closer inspection it seems there's no realistic way to do that. A 5 bit op code is fine, though, and it leaves us plenty of room to work with. I'm assuming at this stage that all of our instruction formats will be an integer number of bytes, so we still have room in our instruction formats if we need to extend that further without pushing anything over the byte threshold.

We've opted to go with an 8-bit immediate in our main instruction type. Our processor is required to handle 16-bit immediates, so this will allow us to load a big immediate in only two steps. If we find we have extra room later, we can always allocate more room to the immediate-- right now we have 3 extra bits, for example. I'm apprehensive to do that right now because we don't know if we'll need those bits later, but I would be very unhappy if we found we had to push our immediate smaller than 8 bits; at that point I think I'd rather just make the instruction format 3 bytes large.

We decided on our procedure call conventions, but they're largely on probation. We said we'll back the arguments onto the stack (in the order listed) and return the value in the accumulator, only because this seemed the easiest way to do it. Perhaps it's not future proof, and perhaps we need to change it in the future, but it seems to work for relPrime and a few other procedures I made up off the top of my head, so I'm satisfied right now.

September 29th, 2018 -- 90 minutes

Once again, no individual work was performed. We are still trying to flesh out the instruction set architecture, which is very much a group effort.

For how long this meeting was, I'd say it was surprisingly unproductive. There were a lot of fruitless arguments. We did still decide on a number of important decisions, however.

We've decided that there will be three separate comparison instructions. Similar to ARM, these instructions will update a special comparison register, and this register will be used in a "jump on condition" command. I'm the one that proposed this partially because of my own biases; I do not like how MIPS requires you to store the result of comparison into an actual, usable register. The other (more vital) reason is that we simply don't have anywhere else to put it. We could store the result of a comparison onto the stack (and, indeed, that possibility was considered), but to me that seems awful for so many reasons. How would it even work? Not beautifully, at least in my mind. The dedicated register will work very well.

We also discussed how we intend to perform jumps, and ended up with, tentatively, four separate commands: one to jump to an immediate, one to jump to an address in the accumulator, one that works pc-relative, and one that jumps to a register (so, ra). We reason that if one of these is unnecessary, we can always remove it later, and if we need more, we can always add them later. This seems like a good enough start for jump commands, and should be enough to work out relPrime.

We ended this meeting by assigning proper individual work for the first time. My task is to create the assembly code fragments that show off how common operations would look in our language. Look forward to that in the next journal entry.

September 30th, 2018 -- 30 minutes

This morning I worked individually on the assembly code fragments. I started by writing the most basic stuff: putting a number into the accumulator, adding two numbers, loading a big immediate-- the basics.

Then I tried something a little fancier: a C classic. Add up the numbers from 1 to 10 using a for loop. The result should be 1+2+3+...+10 = 55. What I found is that it's literally impossible with our current instruction set. You cannot manage both the loop index and the running total at the same time. Even with the stack, there is no way to manage the two variables and end without leaving a giant stack frame behind.

Of course, that problem reaches far and wide. Next I considered a basic procedure call; I would call a procedure, and it would add a and b and return the total. Easiest procedure ever. Per our procedure calling conventions, I put a onto the stack, then I put b onto the stack, then I jump to the function. I pop a off the stack and into the accumulator, and then... what do I do now? There's no way to add a number to the accumulator from the stack. It's impossible. I would need a new instruction, some sort of "stack add"-- sadd, which is exactly how I felt upon this discovery. If we add a sadd instruction, won't we have to do the same for every possible operation? ssub, sorr, sand... I did the math, and it would multiply our instruction count by approximately 1.5. I decided at this point that our architecture was very flawed, and, acknowledging that the instruction set was bound to change much at our next meeting, I decided to put the assembly code fragments on hold for now.

September 30th, 2018 -- 105 minutes

The very same day, but this time in the afternoon. This is the first meeting Ben was able to join us for, and he brought a number of interesting ideas (and arguments) to the table. We already knew going into this meeting that we would meet with Sid and interrogate him on several matters, so we actually solved very little, putting off all proper decision-making until "after we ask Sid." That said, I feel it would be

wrong not to detail the lengthy (oh so lengthy) discussion I shared with Ben over what I will refer to as the Two Accumulator Conundrum, or TAC.

It began when I brought up the issue I discovered when writing the fragments to the group. I found that there was no way to add two variables together, because we could not operate directly off of the stack. The first thought was that we could add an instruction to add off of the stack, but once I mentioned how that would affect our total instruction count we moved away from that idea quickly. Then I proposed my best solution: have multiple accumulators.

The idea is this: We would have, say, 4 accumulators (represented by 2 bits in the instruction's machine code), and they would be indexed, 0-3. Any command that interacts with an accumulator would take an additional argument that indicates which accumulator it operates on. I explained how this would allow us to do any computation we needed by juggling between these accumulators and the stack, but I also noted how this is bringing us further from accumulator and closer to load-store territory. (We are trying very hard to be as non-load-storey as possible.) I mentioned that with only 2 accumulators and enough stack juggling, it is possible to perform any computation. With only 2 accumulators, it's less load-storey, but having to refer to the accumulators by index sort of feels like it's not in the spirit of accumulator. That's the only real mark against this idea; it's efficient, it's really easy to write code for, but it's a little too much like MIPS and we don't like that.

So Ben proposed another idea. We would have two accumulators. Anything that puts values into the accumulators would first check if the first accumulator is empty. If it is, it would operate on that one. If it isn't, it would instead go to the second accumulator. Then, if an add command was issued, it would implicitly add the two accumulators together. This way, you could have two accumulators and never refer to either of them by index. I thought it was a really interesting and unique idea, but I was concerned that it wouldn't actually be able to solve the problem, so I challenged him to a duel. We would each write assembly to solve the 1-10 for loop problem. I already knew I could do it with my idea; after all, my idea was basically MIPS. I wanted to see if we could do it with his.

I finished my code in about two minutes. It was very straightforward, easy to follow, and efficient. I thought if we tried to make a processor with this it would probably go very well, but I was still concerned that it was too much like MIPS. Ben took awhile longer, mainly because this was the first meeting he was able to attend (he'd been out of town for a bit) and he wasn't familiar with the instruction set. After he was done, I found his code confusing and substantially more difficult to follow, but very cool and very unique. I liked that, and I was satisfied that it would be able to do anything we needed it to do, but I was concerned that writing code for it would be a nightmare-- you'd have to keep track of the accumulators and the implicit operations and everything and it seemed so much more difficult. We spent a grand long while discussing the pros and cons of each, and decided this: My idea was better, but it was lame. His idea was worse, but it was cool. We were hesitant to commit to anything until after we'd met with Sid.

October 1st, 2018 -- 15 minutes

This was a very brief group meeting to talk with Sid. I don't have many thoughts to express here, but there is one major thing I should note. I told Sid about the conundrum facing us with the two-accumulator ideas-- mine, where each accumulator is indexed, and Ben's, where the operations are automatic. Sid liked both of these, but proposed a third option: two accumulators, but the second one's only a backup, and you can swap between the two and perform operations across them at will. I mention this here because it is the idea we end up choosing at our next meeting later this night.

October 1st, 2018 -- 135 minutes

This was our longest group meeting to date, but thankfully we didn't spent over an hour discussing which direction to take our architecture. We very quickly decided to go with Sid's idea of a main accumulator and a backup. We decided that it was a good compromise between my solution, which was explicit and efficient, and Ben's solution, which was less load-storey and more interesting. It strikes a satisfying middle ground, and our entire group seems very satisfied with it.

Recall that our processor is named Frankie, after Frankenstein's Monster. The author of Frankenstein is Mary Shelley, which is what we decided to name the accumulators after. the Main became Mary, and the Secondary became Shelley. We also decided that our assembler's name would be Victor, since Victor Frankenstein "assembled" the monster, and the instruction set architecture would be named Clerval, because Henry Clerval was Victor's best friend. If you can't tell, we've become very committed to this theme.

As for actual work: We ended up removing the distinction between instruction types. Now we only have one, a 16-bit instruction that takes a 5 bit op code, a 1 bit flag bit, and an 8 bit immediate (and 2 bits of wasted space-- for now!!). The flag bit decides whether the instruction will operate on an immediate or on the two accumulators. It's what allows us to interact with the second accumulator. In a way, it's no different than having a 6 bit op code, so really, we did end up doubling the number of instructions we have in order to solve this problem, but we weren't using those bits for anything else so personally I'm fine with it.

We also decided to rework procedure calling conventions so that arguments are put into the accumulators instead of onto the stack (unless there are more than two). And we had to add a couple instructions to handle jumping to and from procedures. There were also a couple other edge cases outlined, mainly involving the flag bit and when we should and shouldn't make use of it, but those interactions ended up being fairly obvious.

We spent the late parts of this meeting splitting up and continuing on the deliverables for this milestone. Assembly fragments were torn away from me and passed to Ben instead, and I started work on proper documentation. Eventually we parted, promising to have our individual parts complete by the next time we met.

October 1st, 2018 -- 1 hour

I continued work on the documentation. I like to think I was fairly thorough. I provided information on every register, every procedure calling convention, every instruction-- I even provided an example use for every single instruction that exists right now. I tried to put as much detail into the documentation as I could, because I think it's probably correct to say that I have the best understanding of our instruction set out of anyone right now, and since I wasn't given the task of writing any of the actual assembly I wanted to at least do what I could to pass on my knowledge. I ended up finishing both the documentation and this journal the night after our meeting. Thus my part of Milestone 1 is officially concluded.

October 5th, 2018 -- 45 minutes

The big decision made here is that we're apparently not doing pipelining. I'd always assumed we would because it's so much faster, but apparently Maura, Joy, and Ben have all heard from separate people that it's an absolute nightmare to actually implement, and given I'm not exactly a hardware genius I'm alright with turning down that particular challenge. We're still going to do multicycle, though. Other than that, we wrote out RTL for aadd, then split up the remaining commands to do tonight. We'll meet again tomorrow to discuss them.

October 6th, 2018 (morning) -- 30 minutes

I wrote up the RTL for my six commands. It didn't take long, and no very meaningful decisions were made. Not too much to say here, sadly.

October 6th, 2018 (evening) -- 90 minutes

We discussed our various RTLs and came up with some conventions so they'd be somewhat more consistent. Some examples: writing pc += 2 instead of pc = pc + 1, not capitalizing mary and shelley in the RTL, etc. Not too interesting, but worth noting, at least. We also started on the state diagrams, and

made the shopping list for parts and control bits. We also made a very rough draft of the overall datapath; we don't need it for this milestone, but we figured it'd help us refine some of the RTL if we knew what the datapath was gonna look like.

October 7th, 2018 -- 60 minutes

We met as a group, but ended up working on mostly separate things. We decided to turn the flag bit into an @ sign for easier readability, so I spent a good chunk of time going through all the examples and changing them to be compliant with that. We also made a big state diagram to summarize the generally correct control bits for each major category of instruction. It isn't meant to represent every instruction exactly, but it should give a good idea of what each one will look like.

October 9th, 2018 -- 45 minutes

Super straightforward day. We wrote up tests for the RTL (one for each major category), added a "Recent Changes" section right below the table of contents, and polished up a couple things with what time we had left. The milestone is due today and I'm not very happy with the RTL we have, but that's just how it goes sometimes. I think this milestone as a whole was pretty... dull? The first milestone, we got to make all the fun and interesting decisions, but this one feels more like grunt work. Necessary grunt work, but grunt work nonetheless.

October 10th, 2018 -- 30 minutes

Met with Sid to discuss Milestone 2. Overall he seemed pretty happy with what we've managed to accomplish. Our RTL is good. He said we don't need a Register File, which, in retrospect, is 100% correct. It seems so obvious now, but who knows if we'd have thought of it on our own?

October 13th, 2018 -- 85 minutes

Though we still had our ALUs from 132, I wanted practice with Verilog for this milestone, so I remade the ALU in Verilog. I also made tests. It appears to work.

October 15th, 2018 (afternoon) -- 45 minutes

In class group meeting, though most of the work was done individually. We've started building the remaining major components in Verilog. I've been assigned to create the main memory. I spent most of the meeting time researching how memory is implemented in Verilog. Simple implementations appear to just be an array of registers. I see no great reason to stray from that. If it works for the Internet, it'll work for us!

October 15th, 2018 (evening) -- 45 minutes

I worked on my own in the evening to add support for some additional operations to the Verilog ALU. The one I wrote originally only supported AND, OR, Add, Sub, and setLessThan. I added support for setGreaterThan and setEqualTo, as well as a matching set of tests.

October 16th, 2018 (afternoon) -- 45 minutes

Another in class group meeting. I finished up main memory and helped the other group members with a couple minor Verilog things. Overall, it was productive, but fairly uneventful.

October 17th, 2018 (afternoon) -- 95 minutes

Last in class meeting of Milestone 3. We fixed up the datapath and control signals, and finished testing for all components. Mostly just fixing stuff up. I believe milestone 3 should now be complete. Fall break in the middle was a little annoying, but I think we're caught up, so I suppose all's well.

October 19th, 2018 (afternoon) -- 30 minutes

Our weekly meeting with Sid. I'd say this is the first one that's gone poorly. Our design document wasn't updated in the repo, our integration plan was apparently quite poor, and we were still missing some stuff from as far back as Milestone 1 with the machine code translations. On a more positive note, our control signal specification is apparently very good.

October 20th, 2018 (evening) -- 90 minutes

This meeting was primarily focused on fixing what went wrong in Milestone 3. Ben worked on the long overdue machine code translations while the rest of the group brought the integration testing plan, input/output, and component specifications up to date. Once that was taken care of, we started looking at Milestone 4, but we've yet to delve too far into it.

October 21st, 2018 (morning) -- 60 minutes

Maura's the one responsible for programming the actual control unit in Verilog. In order to do that, she'll need the specifications for the control bits. Since we're doing multicycle, this will naturally take the form of a finite state machine. But given how many states we're going to have to represent, a bubble diagram's not especially practical-- it'd be far too large. Instead, when I made the finite state machine, I opted for a plaintext approach similar to our RTL specification: list each instruction, list the control signals set at any given cycle. It should make it much easier to follow along and program in Verilog.

October 22nd, 2018 (evening) -- 90 minutes

Group meeting again. We hashed our how we're going to handle interrupts. There will be four possible causes: one for input, and three for overflow exceptions (pc, sp, and alu). We will check for input interrupts at the start of each instruction, and we will check for overflow exception interrupts every cycle. We also worked out that freezing the processor is equivalent to setting all control bits to 0, and that the coprocessor would have to keep track of the old PC, Mary, and Shelley, as well as the cause of the interrupt. Once we worked all that out, we moved on to working on integration and the control unit, which are, I think, fairly dry. We're still struggling with Verilog, unfortunately.

October 23rd, 2018 (evening) -- 90 minutes

We picked up where we left off last night: integration and the control unit. We ran into Verilog issues with both. The control unit test suddenly started failing, so this was (after much distress) solved by making a new Verilog testbench. We still have no idea what caused the problem in the first place. We also ran into an issue making the PC block because apparently the output of a submodule in Verilog cannot be a register. We ended up having to remake our register module to resolve this. Between helping everyone with errors, I made very little progress on the ALU block (the integration block I'm responsible for). Mildly unfortunate.

October 24th, 2018 (morning) -- 25 minutes

I realized this morning that the finite state machine still wasn't fully specified on the design document. Well, it is now.