Design Process Journal

Meeting #1 – First the type of Processor that everyone wanted to build was discussed. The vote was between the accumulator and the stack, with the accumulator winning. The accumulator was chosen with the reason that it could potentially be more user friendly. Git Hub was also agreed to be used as the group repo. The next topic was what everyone wanted to focus on when making the processor. It was agreed that we all wanted to make the processor fast in speed but also user friendly. We then discussed potential types and instructions that we would be implementing.

Meeting #2 – The first item discussed in this meeting was the types and there bit usage breakdowns. I, R, and IR types were agreed upon except we were not able to fully agree upon a bit distribution for the R-type. The I-type has 12 bits for the immediate (which we thought would be more than enough for add’s, and’s, or’s, etc.) and any command with I type will be using the main accumulator register. So for example, the I-type add will add an immediate value to what is currently in the accumulator. The IR-type has access to any of the registers and to a 9 bit immediate value. Our ideas for this IR-type included being able to do comparison with the main accumulator to a register (for something like a beq) and then going to an immediate value destination. We then listed out our commands that we wanted and gave them operation codes. We also started working on how some commands would be implemented in assembly code. We also agreed upon using 8 registers including the main accumulator register.

Meeting #3 – The R-type bit distribution has been decided upon. The final R-type has a func code with 2 registers and a tailing 2 unused bits. Our thoughts here, included being able to add, and, or, etc. two registers together and place the value into the accumulator. The func codes were the next item discussed and agreed upon. We have gone through each operation and wrote out how we would like them written in assembly code. Much of the documentation was done during this meeting because we now have everyone agreeing upon the current design of each command and type.

Meeting #4 – The code for Euclid’s algorithm written in our assembly code is almost complete. There has been a key point, raised when trying to do branches using our accumulator. It has been decided that we will create two more operations in the func codes, beqz and bnez. These are branches will compare values against zero. The idea behind these comparisons is that there will be much less swapping of values into registers if we can just check to see if the value in register is zero or not.

Meeting #5 – Tuesday 1/20/15

Drawing out a simplified version of our potential data path, to determine a RTL.

We have agreed on multi-cycle for our processor, one main reason being we will be using the common swap frequently when writing assembly code, multi-cycle should make these swaps take hardly anytime.

Li and lui are going to be hard wired in, saves one instruction for our design.

Copy and move are going to be hard wired and swap will a pseudo instruction.

Josh did the documentation for this meeting, Ben, Jeff, and Thomas drew the design of RTL and the data path on the white board.

Meeting #6 – Wednesday 1/21/15

Ben and Josh only available for this meeting. Listing out the components needed for the agreed upon data path. Described about three fourths of the components (what they do). In this meeting we had a picture of our data path and our RTL, and we went through a listed out each component and control signal we would need. Then we on the white board and in our design journal gave a good description of each component and its purpose. This will continue into the next meeting.

Meeting #7 – Thursday 1/22/15

NOTE DECISION: Each write to register instruction will be written to the main register, so there will be no address needed for a write address.

We have decided to keep the $at register even though as of right now we only have one pseudo instruction, we will soon be adding pseudo instructions, the amount depends on how extreme our final data path is.

Ben and Josh working on the rest of the components list descriptions, with Ben editing the list and Josh doing the documenting for the design process journal.

Thomas working on the tests for RTL.

Jeff will be checking our work over because he was not able to make this meeting. He will also be writing the memo for Milestone II.

Thomas and Josh have begun finalizing the RTL.

Ben is working on the input, output, and control signals.

Branches/Jumps are added to the data path.

NOTE DECISION: We are using an extra adder in the branch because we can do the addition of PC + Immediate while we do the comparison, which saves us a cycle.

Meeting #8 – Sunday 1/25/2015

The Data Path is drawn out in power point to use as a reference for the state diagram.

The State Diagram is being created.

NOTE DECISION: We decided to use a single memory for instruction memory and data memory on our data path instead of splitting it.

NOTE: We are not sure which cycle the control goes into, so for now we are ignoring it and will alter the state diagram later if we need to once we know where the control has to go. We decided to split up work on the components.

We updated the RTL to included load word/store word.

State Diagram is also created in power point.

Thomas, Jeff, and Ben are doing the draw.

Josh is checking and doing the documentation.

Meeting #9 – Monday 1/26/2015

The State Diagram is finalized in the Official Design Documentation.

Verilog is started by Thomas, Jeff and Josh review and continue to work on this.

Josh is adding the files together for the design document.

NOTE DECISION: We will have a separate state for jump and link and for copy.

Thomas is looking into the book to understand the ALU better.

Ben and Josh are working with the new documents, as well as fixing Thomas’s Git Hub.

NOTE: Project in Xilinx is being made on Thomas’s computer.

Meeting #9 – Wednesday 1/28/15

NOTE DECISION: JR-Type now has new bit assignments, the design is a little different but keeping with the consistency of having the op code and register is more important.

Thomas: Updated data path, removed pseudo instructions that were documented as regular instructions. Adder in Verilog created (16 bits).

Jeff: Worked on the controller in Verilog

Josh: Worked with updating documentation. Researched ideas for creating a compiler for our processor.

Ben: Writing tests against the Register Transfer Language in python.

Meeting #10 – Thursday 1/29/15

Ben: Integration tests and testing of the components

Josh: Design Process Journal

Jeff: Memo and description of control signals

Thomas: Narrative explanation of the components

**Strategy for Creating Tests**

- Use Verilog test benches to test each operation

For example (ALU)

- In Verilog initialize everything needed for ALU.

- List out all op codes that the ALU can perform.

- Have an additional wire in Verilog called the expected value.

- For example, perform the addition operation, pick two values that the result is known and see if the result is the same as the expected value.

- Do this for each operation until confident that the operations all work so no more testing will be needed, that way we can use the component confidently in our data path.

**Errors While Testing**

- In Verilog most of the errors that occurred while testing, were caused by a bad test. A handful of the test cases were calculated wrong.

- These errors were simply fixed by fixing the test cases. (most of the time the expected value was incorrect)

**Choice in Architecture Affecting Data Path Design**

- One major effect of our accumulator design on our data path is the need for a zero constant which is used to easily address the main register, without wasting bits in our instructions.

- Another major effect is the need for a main register out (from the register file) to go into the write data, this is because the result of operations will be placed in to the main register, which means it only makes sense to have from the register file, main register into write data.