**Design Journal**

We have decided to make a load-store architectural design for the sake of simplicity. We decided on four bit OP codes, which limits the number of possible instructions to 16, but allows us more bits in the instructions for registers and immediate values.

**Instruction Formats:**

We implemented four instruction types: R-Type, I-Type, J-Type, and A-Type. R-Type instructions receive two source registers (rs and rt) and store the result in a destination register (rd). I-Type instructions receive one source register (rs) and one immediate value, and stores the result in a destination register (rd). J-Type instructions use a 12 bit immediate value as an address to jump to. A-Type instructions receive an 8 bit immediate value and store the result in the destination register (rd).

**Choosing the Registers:**

We first decided which registers were absolutely required for this project. We determined that we would need a constant zero ($0), a return register ($v0), a stack pointer ($sp), an OS kernel register ($k0), two argument registers ($a0, $a1), and a return address register ($ra).

Since we plan on supporting 16 registers, there are nine registers remaining to allocate between s- and t-registers. We decided to allocate five registers as s-registers and four as t-registers. We thought it would be best to have an even balance of s- and t-registers, but chose to have more s-registers since it would be better to be able to save more values if needed.

**Instructions:**

We selected a 4 bit OP code for the instruction types, which limits us to 16 instructions. We selected the basic instructions that are required for Euclid’s algorithm, and added additional instructions that are required for loads, stores, and jumps. We decided on the following instructions for each instruction type:

R-Type: OP(4), rs(4), rt(4), rd(4)

* Add, And, Beq, Bne, Or, Slt, Subtract

I-Type: OP(4), rs(4), rd(4), Imm(4)

* Lw, Sll, Srl, Sra, Sw

J-Type: OP(4), Imm(12)

* J, Jal

A-Type: OP(4), rd(4), Imm(8)

* Addi, Jr

We created an RTL specification for our processor. Along with this, we have updated the component and signals lists and all of this was added to the design document. We've made tests to validate our RTL specification, which seems to work well right now. Soon, we will be able to actually test our design, as we start implementing it in Xilinx. We included a C register in the RTL to comply with our instruction type spec, which works our really well in the RTL document.

Milestone 3:

Our strategy for creating test cases was to test in a manner that went for the extreme values and a normal case. By doing this all ranges of values should work. In the case of the ALU Control lookup just test all values as there is only 16 different ways.

Our datapath revolves around our choice of register to register architecture. We also chose to only have one memory and one ALU. This caused us to have to have several muxs to allow to use the same components multiple times in the same command.

Our architecture caused a few issues with our datapath designed that caused us to have to add extra control signals and make the instruction take longer. The instruction was for branches as we have the [3:0] of the instruction for which register’s value we should add to the program counter. We first run the PC+Register store it into ALUOut then turn off ALUOut write and do the comparison between register 1 and 2. Also for this had to add another input into PCSource from ALUOut.