The Code Hopper

Load-Store

Team 1A

Adam Finer, Runzhi Yang, Katrina Kerrick, Fred Zhang

Contents

Executive Summary

Our group created a multicycle, load-store processor that is capable of running Euclid's algorithm. Taking tips from how accumulator-based processors dictate assembly, every instruction that doesn't use immediates takes only a maximum of two registers as parameters, instead of three. This extra space is used for a condition code, which dictates whether the instruction should be executed or not. With load-store, our assembly language is very concise and powerful, and programs can be written in much fewer lines than other architecture types. This processor is capable of function calls, recursion, and uses two stacks to keep track of variables and data. It is powerful, fast, and we are proud of it.

Introduction

Our processor uses a multicycle datapath. It combines a program counter, block memory, register file, ALU, and many, many muxes and small registers to process instructions and programs. The strength of this processor lies in its ability to skip instructions. Every instruction that doesn't use immediates takes a condition code, and only if this code is valid does the instruction execute. This allows for easy multi-line if statements, loops, and large if/else-if/else statements. Our original objective was to make the processor very easy on the hardware side to design, but we ended up trying to create a very powerful and efficient instruction set, both in cycle time and number of lines of code. The processor is capable of interrupts and exceptions, as well as I/O. It does not have any major weaknesses, but there are many places where it could be improved, like making the stack pointer accessible to the user.

Body

After deciding on a load-store architecture, coming up with how to write instructions was a lot simpler. Though our processor is not an accumulator, our instruction set was designed so that it could be used on an accumulator architecture. Every operation instruction takes the first register passed in and uses it also as the register that is written to afterward. In programming terms, this makes every "+" act like a "+="—every time an addition is made, the result is stored back in the first register. This saves a lot of space in the instructions and allows for use of the flag register. The flag register stores a three bit value that shows whether one number is less than, equal to, or greater than another number. These two numbers are supplied by the programmer and must be kept track of manually, but the value in the flag register is never zero. The flag register can be updated with the condition code from compare instruction (cmp). For every instruction that doesn't take an immediate (any non-I-type instruction), the three of the four saved bits are reserved for the condition code. If the condition code anded with the flag register isn't zero, then the instruction is executed. This means that if statements can be performed without using branches, saving many lines of code when writing long problems. However, because our jumps are not I-type instructions, they also take a condition code. In this processor, jumps and branches are the same thing. After a very long debate we chose to scrap

all branches and get rid of unconditional jumps, instead having two different types of jumps—jump label and jump register. This allows for both the functionality of jumps and branches. In addition, with conditional jumps and the compare command, programmers for our processor have no need for a zero register or a one register, though this was a much-debated topic.

This processor does not do shifts. Though our group left the opportunity for a shift command open in the case that we finished our other designs early, it was decided that it would be too difficult to write one in.

The CPU has two components--the datapath and the control. The datapath processes data and manipulates data flow, while the control instructs the datapath what job to do. In our multi cycle processor, the datapath is designed to be synchronous. The Control Unit is a mealy state machine, able to output different control signals in the same state based on the datapath status flag.

Memory and the register file are clocked so they only change their output synchronously on the clock edge. In addition, block memory only takes one cycle to use. Most inputs to these components are controlled by control signals and muxes. There are a few special registers--PC keeps track of program counter, which increments by one unless there is a jump instruction; SP keeps track of the stack pointer and only push and pop can change it by one; and the flag keeps track of the last comparison result and enables our processor to skip instructions and jump conditionally. The datapath has three status flags to inform the control of its status--Op specifies which instruction is executing, perform decided whether this instruction will be finished, and LMC controls whether to load from memory for that instruction.

The control unit has eight total states. All instructions execute in 2 to 4 cycles. For most instructions, there is instruction fetch, decode, memory load, and execution. The briliand design enables our control unit to branch out at the second cycle and save one cycle. For simple instructions like cpi and lui, control manages to skip the decode cycle and write the immediate value into the register in the second cycle. Those two instructions only takes two cycles, then. For non-I-type instructions, when Perform is low, the control changes to state 1 and skips the instruction. Skipping takes only two cycles. For a non-I-type instruction when LMC is high, the control goes to a special load memory cycle and continues normally afterwards. We manage to combine the write back cycle with execution, so the executed result is written into a register or memory in the same cycle.

Our processor can handle I/O and exceptions. Whenever there is an I/O interrupt, we can RESET our CPU to a designated PC address, and guide it to execute instructions there. In the meantime, the stack pointer is reset back to the top of stack and register $t0 contains an argument input from the user. Whenever there is an exception, the exception bits go high, RESET the CPU and abort the process.

Most of our Xilinx models are implemented with Verilog. WIth Verilog, building a CPU is really easy and straightforward.

Muxes can be built with ternary operator in Verilog.

The instruction register is just a 16-bit latch outputting different portions of its values, which can be specified with the { , } syntax. Writing into a latch can be implemented with an always @(*) block.

The register file is just a group of 16 16-bit registers. To implement this with Verilog, we can directly declare an array of 16 registers, and update outputs and the clock edge. Always @(posedge CLK) is very handy. One tricky part is to use non-blocking assignment <= to shrink the clock cycle.

The ALU was implemented with Verilog with cases, which generate a mux.

Memory is a bit tricky. We used the block memory provided by Xilinx IP generator. In order to make the fastest processor, the block memory would need a whole clock cycle to read and write. It only outputs data at the clock edge. We made a few modification late in the term to accommodate this memory design. A delayMux is added and instruction register becomes a latch. Those design decision and changes are discussed in details in design document and design journal.

The Flag was implemented with Verilog. It has three flip flops, combinational logic to determine Perform, its output. Perform is asynchronous, so it updates whenever a new condition code arrives.

Our Control Unit is implemented in Verilog with both always @(posedge CLK) blocks to control state transition and continuously assign the control signal output. Since the datapath is much slower than the control, and the state transition is not part of the critical path of our CPU, there is no need for optimization. However, our datapath needs to wait for the control signal to start performing its work, so control signals are part of the critical path. Control signals are related to both state and input. K-Maps and utilizing the ability to ignore a signal simplified the control signal logic greatly.

As for I/O and exception handling, our processor has the ability to generate exceptions, like segment fault, overflow, or I/O interrupts by putting arguments into the register upon pushing a button. We did not build exception handlers for them yet, though, and all exceptions just abort the process. I/O is working with the board. We can send an argument to the $t0 register and display value in $v0 register with LCD.

For the testing of our processor, we wrote several different levels of testing—unit testing, two levels of integration testing, and system tests.

Our unit tests are fairly straightforward. For every component that we wrote, we made a list of desired inputs and expected outputs that this component could provide while it was isolated from the rest of the datapath. In addition to a base case that covered general instructions, each component would have all of its corner and edge cases tested so that our group would know that it could take strange instructions and wouldn't break seemingly randomly in the middle of a program.

The unit test for control unit requires a lot of different cases. To make our life easier, we create a ControlUnitTester, which is built with all cases. Regarding each state and datapath flag, it provides necessary information, including whether certain control bit is 1, 0 or don't care, what the next state is supposed to be, whether combination of state and status flag are unreachable. ControlUnitTester copies specifications directly from our design and cover all the cases. It is very slow, but guaranteed to match the design. Then we can run our control unit with all combination of possible inputs several cycles and compare outcome with ControlUnitTester. A lot of buts are caught in this process, and there is no more bugs discovered in Control Unit after unit test.

Integration tests were more difficult. First, we had to come up with meaningful combinations of the units of our datapath. As a group, we sat down and picked useful combinations and voted which ones to keep and which to discard. After that, we came up with useful combinations of those integration tests to be additional integration tests. Then, for each combination, we came up with a list of inputs and outputs to test base cases and all edge and corner cases. These tests were more difficult to write, as we had to test components that we had not written. Memory and the register file were difficult because they first had to be loaded with values, then have the values read. In addition, the clock had to be cycled manually during these tests, and components like multiplexers had to be tested in a certain order.

System tests are composed of several different levels. The first level of system tests is for individual instructions. We tested each instruction in a base case and all edge cases possible, causing exceptions like overflow or segfaults if possible. Once each instruction was verified to be working, small sets of instructions were tested together. These tests were composed of our short code snippets from the first milestone of the project and pseudoinstructions that compiled down into more than one regular assembly instruction. These did not need edge cases to be tested, though we were careful to use plenty of tests to help make us more certain of the abilities of our processor. Finally, we tested the Euclid's algorithm program on our processor and debugged that to prove that long programs could run without problems. Most of the problems we ran into with making Euclid's algorithm work properly were with Xilinx, and not with our processor. However, we were forced to make a small change to our instruction set to make the program run easier, which was adding cpi (copy immediate) to the standard instruction set instead of having it be a psuedoinstruction.

After these tests all ran and passed, we felt very confident in saying that our processor was functional.

Our performance data can be summed up in eight points and a table:

1. The total number of bytes required to store both Euclid's algorithm and relPrime as well as any memory variables or constants:

The main function has 3 instructions, 10 Bytes, including one jl instruction at the end to create an infinite loop. RelPrime has 19 instruction, 42 Bytes. Gcd has 13 instructions, 30 Bytes. Altogether we have total 33 instructions, so an 82 Byte program.

We only need 5 slots in the memory for the backup register. The minimum memory run for Euclid's algorithm is 82 + 5 * 2 = 92 Bytes.

2. The total number of instructions executed when relPrime is called with 0x13B0 (the result should be0x000B using the algorithm specified in the project specifications).

   51090 instructions

3. The total number of cycles required to execute relPrime under the same conditions as Step 2.

   143029 cycles

4. The average cycles per instruction based on the data collected in Steps 2 and 3.

   2.7995 CPI

5. The cycle time for our design is

   10.770ns

6. The total execution time for relPrime under the same conditions is

   143029 cycles * 10.770ns/cycle = 1.5404ms

7. The device utilization summary:

Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Flip Flops	21	9,312	1%	
Number of 4 input LUTs	71	9,312	1%	
Number of occupied Slices	36	4,656	1%	
Number of Slices containing only related logic	36	36	100%	

Number of Slices containing unrelated logic	0	36	0%	
Total Number of 4 input LUTs	71	9,312	1%	
Number of bonded IOBs	39	232	16%	
Number of BUFGMUXs	1	24	4%	
Average Fanout of Non-Clock Nets	3.20			

Conclusion

In conclusion, our design came out exactly how we wanted it to. Our group is extremely proud of the processor we've created, and together we debated long and hard about which features to keep and which to discard. However, we agreed that we should do a multicycle processor, and while our assembly language was heavily influenced by ideas from the accumulator architecture, we decided that load-store would make a more feasible processor. Once we focused on creating an excellent assembly language, the rest of our ideas aligned and we were able to make streamlined decisions. The great strength of our processor are the condition codes that accompany most instructions and the flag register that deals with them, allowing for easy skipping of instructions without a branch statement. Our processor is very developed, able to handle I/O and interruptions, and while there are several improvements we would have liked to make as a group, there are no major weaknesses to our design.

Appendix:

Description of Registers Available

There are sixteen registers total.

There are eight "t" registers, which are temporary variables. These registers can be overwritten inside a function and do not need to be backed up on the stack. However, "t" register value would be blowed away across function call, so we would need to back them up if they are still needed. They are also used as parameters for functions. For example, if there is a function funct1(num0, num1, num2), num0 would be stored in $t0, num1 would be in $t1, and num2 would be in $t2.

There are four “s” registers. These registers must be backed up before use and restored after use inside a function. Its value is preserved across a function call. “s” registers are general purpose and can be used at any time.

There are two “v” registers. These registers function exactly like t registers, in that they don’t need to be backed up and can be used for general purposes. However, whenever a function is called, the v registers will contain the return value of the function. The $v0 register is also the display register. This is the final output number and will be displayed to the user.

There is one “at” register. This register is special use for when the compiler needs to split a pseudo-instruction into multiple instructions. This register may not be used in normal code.

There is one “ra” register. This register is for the return address of a function. When a function is called, jal automatically saves PC counter into the $ra register. The program counter is then changed to the address of the function. $ra is a reserved register and may not be used for general purpose needs. $ra must be backed up on the stack before calling a function and restored after the call is complete.

There is one “dp” register. This register keeps track of the data stack, where we can store variables (see register conventions). $dp functions as a second stack, very similarly to the stack $sp keeps track of. The value of $dp at the beginning and end of a function must match.

Table 1 Register address

REGCODE	00	01	10	11
00	t0	t1	t2	t3
01	t4	t5	t6	t7
10	s0	s1	v0	v1
11	s2	at	ra	dp

English description of each machine language instruction format

R-type -- regular instruction format with two 4-bit register (r1 and r2), 1-bit memory-load-option-code (LM), and a 3-bit cc. Most of the time, instructions of this format will perform general operations with the values in r1 and r2, like add and or, and stores the result into r1. If one wants to use the CC, please refer to the later explanation on cc. If LM is 1, then it would perform the operation with Mem[R[r2]] instead of R[r2], so just go to memory to load value.

op(4)	r1(4)	r2(4)	LM(1)	CC(3)

I-type -- immediate instruction format with one 4-bit register (r1) and 8-bit immediate value (imm). It performs a general operation of values in r1 and imm and stores it into r1.

op(4)	r1(4)	imm(8)

J-Type -- jump instruction format takes up 2 instructions (32 bits). If jop is set to 1 then the second instruction contains a 16 bit address and when it goes to that address, it also links by setting $ra to be PC. If jop is set to 0, then it will only go to that address.

op(4)	$ra	unused(4)	jop(1)	CC(3)

address(16)

Explanation of Condition Code (CC)

All R, B, and J type instructions use a three-bit condition code. This three-bit code specifies one of the following:

Table 2 Conditional Code (CC)

explanation	**CC**	**condition**
greater than	100	>
greater than or equal	110	>=
less than	001	<
less than or equal	011	<=
equal	010	=
not equal	101	!=
always execute	111	True

How condition code work:

There is a three-bit flag register inside our CPU, which stores the last comparison result. The only command that would update flag register is cmp.

```
cmp   $t0   $t1
```

In the above example, cmp would subtract $t0 and $t1 and set flag bits depending on the result. Three bits of flag represent negative (smaller than), zero (equal), positive (bigger than) respectively. Only one bit of those flags can be 1 since $t0 - $t1 can only be negative number, zero or positive number.

Whenever an instruction is executed (except I-type, without CC), we compare those condition bits and flag bits. Similarly, three CC bits represent negative (smaller than), zero (equal), positive (bigger than). As shown in the table above, if the flag bit indicate that the last comparison satisfies the condition, this instruction is going to be executed, otherwise, it is going to be skipped.

This cool feature enables us to do short 'if' statement without branches. We can do a compare and then skip instruction that we do not want to execute. Instruction skipped would take only two cycles, so a pretty efficient way to do single line 'if'.

Take, for example, the following python code:

```
if   $t0  >   $t1 :
      B1
      B2…
elif $t0 == $t1:
      E1
else:
      S1.
```

```
cmp    $t0, $t1
B1     ……  ,  100
B2     ……  ,  100
E1     ……  ,  010
S1     ……  ,  001
```

For short if statements of no more than 2 instructions, CC can save the trouble multiple branches have by simply skipping over certain instructions. For one single line if, this is even better. However, there is one restriction for CC. I-type instruction cannot take advantage of CC.

By default, the condition code is 111 if it is not specified in the instruction, so these instructions are always going to be executed.

English Description of Each Instruction and Semantics

Real Instructions:

add -- adds a register and either a value from another register or a value from memory together and stores the value in the first register. This is an R-type instruction. All R-type can have optional condition code at its end.

add $r1, $r2 add $r1, ($r2) add $r1, $r2, 110

addi -- adds a register to an immediate value and stores the result in the register. This is an I-type instruction.

addi $r1, imm (immediate is 8 bits)

sto -- takes the value in a register and then stores it in memory at a specified address. That address is either from a register or a location in memory. This is an R-type instruction.

sto $r1, $r2 sto $r1, ($r2) sto $r1, $r2, 100

lui -- takes the immediate value and places it in the upper portion of whatever register specified. This is an I-type instruction.

lui $r1, 8-bit immediate

sub -- takes a register and subtracts either a value from another register or memory from it and stores the value in the register. This is an R-type instruction.

sub $r1, $r2 sub $r1, ($r2) sub $r1, $r2, 100

cmp -- takes a register and another register or a value from memory and compares their values. It then sets the three flag bits to show whether the first register is smaller (100), equal (010), or greater (001) than the second. This is an R-type instruction, so it also takes CC. We can optional skip cmp, so we can do multiple comparison in a row.

cmp $r1, $r2 cmp $r1, ($r2) cmp $r1, $r2, 100

cp -- takes one register and copies its value into a different register in the format of destination register, source register. This is an R-type instruction.

cp $r1, $r2 cp $r1, ($r2) cp $r1, $r2, 100

cpi -- initializes a register with an immediate value. This is an I-type instruction.

cpi $r1, imm (immediate is 8 bits)

and -- ands a register and either a value from another register or memory together and stores the value in the first register. This is an R-type instruction.

and $r1, $r2 and $r1, ($r2) and $r1, $r2, 100

xor -- takes a register value and another register or a value from memory, xors them together, and stores the new value in the first register. This is an R-type instruction.

xor $r1, $r2 xor $r1, ($r2) xor $r1, $r2, 100

push -- decrements the stack pointer register and then stores the value in the register in the stack. This is an R-type instruction.

push $r1 push $r1, 100

pop -- loads the value from the stack at the location of the stack pointer into the register and then increments the stack pointer. This is an R-type instruction.

pop $r1 pop $r1, 100

or -- ors a register and either a value from another register or memory together and stores the value in the first register. This is an R-type instruction.

or $r1, $r2 or $r1, ($r2) or $r1, $r2, 100

ori --- ors a value from an register with an 8 bits immediates then puts the value into the register, the immediate value can range from 0 to 256

or $1, imm (immediate is 8 bits)

jr -- takes the value in a register and jumps to that instruction number. This is an R-type instruction.

jr $r1 jr $r1, 010

jl -- jumps conditionally to a label. It has the same opcode as jal, but with a flag bit(LM) = 0 to discriminate them. This is a J-type and takes 4 bytes memory. Condition code would determine whether the jump would be proceeded.

jl label jl label, 010

jal -- jumps conditionally to a label and link. It has the same oncode as jl, but with a flag bit(jop) = 1 to discriminate them. Apart from setting the PC to new address, it also sets $ra to old PC address. jal is usually used to do function calls. This is a J-type and takes 4 bytes memory.

jal label jal label, 001

Pseudo Instructions:

andi -- ands a value from a register and an eight bit immediate and puts the value into the source register.

```
andi    $r1, imm (8 bit immediate)
        cpi     $at, immediate
        and     $r1, $at
```

andi (big) -- ands a value from a register and an eight bit immediate and puts the value into the source register.

```
andi    $r1, imm (16 bit immediate)
        lui     $at, upper immediate
        ori     $at, lower immediate
        and     $r1, $at
```

xori -- xors a value from a register with an eight bit immediate and puts the value in the source register.

```
xori    $r1, imm (8 bit immediate)
        cpi     $at, immediate
        xor     $r1, $at
```

xori (big) -- xoris a value from a register and an eight bit immediate and puts the value into the source register.

```
andi    $r1, imm (16 bit immediate)
        lui     $at, upper immediate
        ori     $at, lower immediate
        xor     $r1, $at
```

clear -- sets a register to have a value of zero.

```
clear   $r1
        cpi     $r1, 0
```

addi (big) -- adds a 16 bit immediate to a value in a register.

```
addi    $r1, imm (16 bit immediate)
        lui     $at, upper immediate
        ori     $at, lower immediate
        add     $r1, $at
```

cpi (big) -- sets a register to a 16 bit immediate value.

```
cpi     $r1, imm (16 bit immediate)
        lui     $r1, upper immediate
        ori     $r1, lower immediate
```

cmpi -- compares a value from a register with an immediate value.

```
cmpi    $r1, imm (8 bit immediate)
        cpi     $at, immediate
```

```
            cmp     $r1, $at
```
cmpi (big) -- compares a value from a register with an immediate value.
```
        cmpi    $r1, imm (16 bit immediate)
            lui     $at, upper immediate
            ori     $at, lower immediate
            cmp     $r1, $at
```
beq -- branches to a label if two registers are equal.
```
        beq     $r1, $r2, label
            cmp     $r1, $r2
            jl      label, 010
```
bne -- branches to a label if two registers are not equal.
```
        bne     $r1, $r2, label
            cmp     $r1, $r2
            jl      label, 101
```
blt -- branches if the first register is less than the second register.
```
        blt     $r1, $r2, label
            cmp     $r1, $r2
            jl      label, 100
```
bgt -- branches if the first register is greater than the second register
```
        bgt     $r1, $r2, label
            cmp     $r1, $r2
            jl      label, 001
```
ble -- branches if the first register is less than or equal to the second register
```
        ble     $r1, $r2, label
            cmp     $r1, $r2
            jl      label, 110
```
bge -- branches if the first register is greater than or equal to the second register
```
        bge     $r1, $r2, label
            cmp     $r1, $r2
            jl      label, 011
```

Explanation of Register Conventions

In general purpose assembly, the t and v registers may be used. However, these values are potentially destroyed whenever a function call is made, so their values must be saved on the stack or into an s register. The t registers are also used as parameters for function calls, starting with the lowest t register first. If more than eight parameters are needed, they are to be stored on the stack. The v registers are used as output from functions.

The s registers may be used in general purpose assembly code as well, but they must be backed up before use and restored after use. In exchange, their value is guaranteed across function calls.

The at register is for the compiler's use only. When pseudo-instructions are changed into a set of regular instructions, if a temporary variable is needed, at is to be used. It is very important that at is not used for general purpose programming.

The ra register is for the return address of a function only. If it is ever changed, it must be backed up first and restored afterward. This allows for the nesting of function calls. For example, if main calls add(), and add() calls sub(), add() must back up the ra value from main and restore it after the sub() call has finished. Additionally, any t or v registers that hold important values must be backed up or their values will be lost.

We separate the regular stack into a data stack and a procedure stack. The procedure stack keeps track of return address and restored 's' and 't' registers while the data stack keeps track of arrays and other data that needs a reference. **No pointer is allowed to point to any address within the procedure stack.** This design prohibits buffer overflow attacks because return address and any array will be in two different stacks and no one can modify return addresses in the procedure stack.

There are two separate stack pointers for data stack and procedure stack -- $dp and $sp.

$dp keeps track of the data stack. Sometimes we need to allocate a big chunk of local memory for an array or object. We can decrement $dp to allocate appropriate amount of data. Then the value of $dp would be the pointer to the data chunk we just allocated. For example, to initialize an array of chars with size of 20 --

```
char[] a = new
char[20];
…
return ....
```

```
addi    $dp,  -20
cp      $t0,   $dp    # $t0 points to the  new array
……
addi    $dp,   20     # free memory before end this
function
```

For other times, we might want a reference to some data, and pass it to another function (similar to the idea of objects). Again, we can decrement $dp to allocate the appropriate amount of memory and then get a pointer to that data. For example, pass reference in C program --

```
char a  =   20;
...
func( &a )
b = b + a
...
return
```

```
cpi     $t1,  20
addi    $dp,  -1
sto     $t1, $dp
cp      $s0,  $dp    # now $s0 is address of variable a
...
cp      $t0, $s0    # pass the address of a as parameter
jal     func
add     $s2, ($s0)  # restore possible changed value of a
...                 # from the data stack.
```

	`addi $dp, 1`

Note that value might have changed during the function call, so it is necessary to restore its value from data stack after the function call. Except global variables, all addressable memory (array, struct, object) created inside a function must be stored in the data stack.

Therefore, all pointers in the program have to point somewhere within data stack or heap.

We have a procedure stack pointer ($sp) register besides those sixteen registers, but not accessible through assembly. It keeps track of the procedure stack, which is only used to backup and restore register for procedure calls. We hide $sp form programmers, so there will never be a pointer to somewhere in procedure stack. Therefore, no programmer will be able to change data in procedure stack. Attacks like buffer overflow can never exist in our processor, because the input data are in the data stack, while the return address is in the procedure stack. Furthermore, to protect that data, our memory is supposed to throw a segment fault if anybody tries to modify data in the procedure stack.

Another advantage of having push and pop is that they are fast. For MIPS, every function call needs to increment and decrement stack pointer and then backup data one by one. Our ‘push’ and ‘pop’ write and read data as well as update the stack pointer, but takes the same amount of time with real memory.

Assembly Language Translation Rule

For each instruction in the Assembly language, first translate the name of the instruction to an opcode using this table. All instruction format except I-type would have 3 bits conditional code (CC) at the end. If not specified with the assembly, those three bits is going to be 111 defaultly.

Table 3 op code

OPCODE	00	01	10	11
00	add [R]	addi [I]	sto [R]	lui [I]
01	sub [R]	cmp [R]	cp [R]	cpi [I]
10	and [R]	xor [R]	push [R]	pop [R]
11	or [R]	ori [I]	jr [R]	jl/jal [J]

After the first four bits are determined, the rest of the instruction is determined based on what type of instruction it is.

If it's an R type:

Translate the two register names into register codes and append them. If the second register had parenthesis around it symbolizing a retrieve from memory, then the next bit (LM) is one. Otherwise, it is zero. The last three bits are condition code (CC) used to decide whether to skip this R-type instruction or not.

There is a special case for jr, which only takes one register argument. The register containing data to jump to specified in r2 instead of r1.

opcode (4) + r1 (4) + r2 (4) + LM (1) + CC(3)

If it's an I Type:

The next four bits are the register address. After that is an eight bit immediate. I-Type is the only type without CC.

opcode (4) + r1 (4) + imm(8)

If it's a J Type:

Its machine code takes 4 bytes instead of 2 bytes as others. The first 2-byte has 4-bit opcode at front and 3-bit CC code at end. If it is jr, then the last fourth bit(LM) is set to be 0. If it's jr, then LM is set to 1 and second register slot(r2[11-8]) should be the register address of $ra to simplify hardware implementation. The second 2-byte contains the new address to jump to.

jl	jal
opcode (4) + unused(8) + 0 + imm(3)	opcode (4) + unused(4) + 1110($ra) + 1 + imm(3)
address(16)	address(16)

Example Assembly Language programs

Euclid's Algorithm

Address	**Assembly code**	**Binary Representation**	**Comments**
0x0000	__relPrime:	`__relPrime:`	
0x0001	push $ra	`1010111000000111`	# store $ra on the stack
0x0002	push $s0	`1010100000000111`	# store old $s0 on the stack
0x0003	push $s1	`1010100100000111`	# store old $s1 on the stack

```
0x0004          cp $s0, $t0          0110100000000111    # put argument to a s register
0x0005          cpi $s1, 2           0111100100000010    # int m = 2;
0x0006  WHILE1:                      WHILE1:
0x0007          cp $t0, $s0          0110000010000111    # put variables into argument registers
0x0008          cp $t1, $s1          0110000110010111
                                     1111000000000111
0x000a          jl __gcd             0000000000010101    # call function gcd
0x000b          addi $s1, 1          0001100100000001    # m = m + 1;
0x000c          cpi  $at, 1          0111110100000001    # check to see if the output is not 1
0x000d          cmp  $v0, $at        0101101011010111
0x000e          jl WHILE1, 101       1111000000000101    # loop if return value  != 1
0x000f                               0000000000000110
0x0010          cp $v0, $s1          0110101010010111    # set retValue = m
0x0011          pop $s1              1011100100000111    # restore previous $s1
0x0012          pop $s0              1011100000000111    # restore previous $s0
0x0013          pop $ra              1011111000000111    # restore previous $ra
0x0014          jr $ra               1110111000000111    # return m
0x0015  __gcd:                       __gcd:
0x0016          cpi   $t7, 0         0111011100000000    # create a zero for comparisons
0x0017          cmp  $t0, $t7        0101000001110111    # check if a == 0
0x0018          cp   $v0, $t1, 010   0110101000010010    # if a == 0, set retValue = b
0x0019          jr     $ra, 010      1110111000000010    # if a == 0, return b
0x001a          cmp  $t1, $t7        0101000101110111    # checks if b !=0
0x001b          jl RET, 010          1111000000000010    # skip the while loop if b == 0
                                     0000000000011000
0x001d  WHILE2:                      WHILE2:
0x001e          cmp  $t0, $t1        0101000000010111    # check if a > b
0x001f          sub  $t0, $t1, 001   0100000000010001    # a = a - b;  if a > b
0x0020          sub  $t1, $t0, 110   0100000100000110    # b = b - a;  if a <= b
0x0021          cmp  $t1, $t7        0101000101110111    # check if b != 0
0x0022          jl    WHILE2, 101    1111000000000101    # loop if b != 0
                                     0000000000011101
0x0024  RET:                         RET:
0x0025          cp $v0, $t0          0110101000000111    # set retValue = a
0x0026          jr $ra               1110111000000111    # return a
```

Assembly Language Fragments for difficult instructions

_clear $t0	0100000000000111
sub $t0, $t0	
_beq $t1, $t2, label	0101000100100111
cmp $t1, $t2	1111000000000010
jl label, 010	(label)
_blt $t1, $t2, label	0101000100100111
cmp $t1, $t2	1111000000000100
jl label, 100	(label)
_ble $t1, $t2, label	0101000100100111
cmp $t1, $t2	1111000000000110
jl label, 110	(label)
_addi(big) $t1, big	0011110100000000
lui $at, upperbig	1101110100000000
ori $at, lowerbig	0000000111010000
add $t1, $at	
_li(big) $t1, 0xeeff	0011000111101110
lui $t1, 0xee	1101000111111111
ori $t1, 0xff	
_cp(big) $t1, (big)	0011000100000000
lui $t1, upperbig	1101000100000000
ori $t1, lowerbig	

RTL Details

For all instruction except I-type, they might be skipped if flag does not match condition code. In cycle 2, flag register compares condition code (CC) and flag bits and output perform bit into control. When perform is 1, it signals that flag bit satisfies CC and this instruction is going to be executed, otherwise, the control is going to skip this instruction and starts the next one.

For most R-type instruction except push and pop, if LM = 1, there is an extra cycle between Cycle 2 and Cycle 3 to load memory. If LM = 0, there will not be LM cycle. All of those magic happen inside control, which has LMC input from datapath. This extra cycle is in the red box in the RLT below. After this extra cycle, everything else works exactly the same.

Table 4 RTL first half

	Calculations	cmp	sto	cp	jr	jl	jal
Cycle 1	IR = Mem[PC] PC += 1						
Cycle 2	A = R[IR[11:8]] B = R[r2]					PC += 1 B = Mem[PC]	
LM cycle	B = Mem[B]						
Cycle 3	R[IR[11:8]] = A op B	flag = sign(A – B)	Mem[B] = A	R[IR[11:8]] = B	PC = B		PC = B R[IR[11:8]] = PC

Table 5 RTL second half

	addi	lui	cpi	ori	push	pop
Cycle 1	IR = Mem[PC] PC += 1					
Cycle 2	A = R[r1] B = R[r2]	R[r1] = Upper(imm)	R[r1] = SignExtend(imm)	A=R[r1]	$sp -= 1	B = Mem[$sp]
Cycle 3	R[r1] = A + Sign(imm)			R[r1]= A| ZeroExtend(imm)	Mem[$sp] = A	R[r1] = B $sp +=1

RTL Testing methods

For all instructions, besides the jumps and branches, make sure pc was incremented by 2 in all cases.

Table 6 RTL Testing

Instruction	Opcode	Testing method	

addi	0001	addi $r0, 5 0001000000000101 Assume $r0= 6. The expected value of $r0 after the execution should be 11	addi $r1, 1 0001000100000001 Assume $r1=0xffff, and adding one causes overflow.		
lui	0011	lui $r0, 0xff 0011000011111111 The expected value of $r0 after the execution should be 1111 1111 0000 0000 regardless of what was previously in $r0	lui $r0, 0xaa 0011000010101010 The least 8 significant bits of the immediate should be loaded.		
cpi	0111	cpi $r1, 0xff 0111000111111111 The expected value of $r1 is 1111 1111 1111 1111	cpi $r1, 0x00 0111000100000000 Expected value of $r1 is xxxx xxxx 0000 0000		
ori	1101	ori $r1, 0xff, the value in r1 is 0x00000000 The expected value of $r1 is 0x1111 1111	ori $r1, 0xf0, the value in r1 is 0xf00f The expected value is 0xf0ff		
sto	0010	sto $r1, $r2 0010000100100111 If $r1 is 0xffff and $r2 = 0x0022, then the memory at 0x0022 will be 0xffff after the execution	sto $r1, ($r2) 0010000100101111 If $r1 is 0xffff and $r2 is 0x0001, then the value at the address specified by what is at 0x0001 in memory will be 0xffff	sto $r1, $r2, 100 0010000100100100 If the cc is true, then the instruction will be executed	
cp	0110	cp $r1, $r2 0110000100100111 if $r2=0x0033 initially, then after	cp $r1, ($r2) 0110000100101111 if $r2=0x0112 and the value at 0x0112 in memory is 12,	cp $r1, $r2, 100 0110000100100100 if the cc is true, then the instruction will be executed, use the	

		the execution, $r1 should be 0x0033	after execution, $r1 should be 12.	initial conditions from the first column	
add	0000	add $r1, $r2 0000000100100111 input: $r1 = 0x0002, $r2= 0x0003 check that $r1 = 0x0005 after execution	add $r1, ($r2) 0000000100101111 input: $r1 = 0x0003, $r2=0x0002 Assume value at 0x0002 in memory = 0x0004 check that $r1=0x0007	add $r1, $r2, 110 0000000100100110 if the flag does not match CC, make sure the add was not done, otherwise do previous. Use initial conditions from first test column.	add $r1, $r2 0000000100100111 $r1 = 0x7fff, $r2 = 0x0001 This causes overflow. Should cause Overflow = 1
sub	0100	sub $r1, $r2 0100000100100111 input: $r1 = 0x0002, $r2= 0x0003 check that $r1 = 0xffff after execution	sub $r1, ($r2) 0100000100101111 input: $r1 = 0x0003, $r2=0x0002 Assume value at 0x0002 in memory = 0x0001 check that $r1=0x0002	sub $r1, $r2, 100 0100000100100100 if the flag does not match CC, make sure the add was not done, otherwise do previous. Use initial conditions from first test column.	sub $r1, $r2 0100000100100111 let $r1 = 0x7fff let $r2 = 0xffff Make sure that it can do negatives properly.

cmp	0101	cmp $r1, $r2
		0101000100100111
		if $r1=0x0002=$r2, check to make sure the value in the flag register equals 010 after the comparison. To check to see if the test ran, run
		add $r1, $r2, 010
		0000000100100010
		and check to make sure that the instruction does change the value in $r1.
			cmp $r1, ($r2)
			0101000100101111
			if $r1=0x0002, $r2=0x0014, and the value at 0x0014 in memory is 0x0001, make sure the flag value is for greater than. To check to see if the test ran, run
			add $r1, $r2, 100
			0000000100100100
			and check to make sure that the instruction does change the value in $r1.
				cmp $r1, $r2, 100
				0101000100100100
				if the flag does not match CC, make sure the value in the flag register was unchanged.
				otherwise do previous.	

and	1000	and $r1, $r2 1000000100100111 input: $r1 = 0x0003, $r2= 0x0005	and $r1, ($r2) 1000000100101111 input: $r1 = 0x0003, $r2=0x0002	and $r1, $r2, 100 1000000100100100 if the flag does not match CC, make	

		check that $r1 = 0x0001 after execution	Assume value at 0x0002 in memory = 0x0004 check that $r1=0x0000	sure the value in $r1 is unchanged, otherwise do previous. Use initial conditions from first test column.	
xor	1001	xor $r1, $r2 1001000100100111 input: $r1 = 0x0003, $r2= 0x0005 check that $r1 = 0x0004 after execution	xor $r1, ($r2) 1001000100101111 input: $r1 = 0x0003, $r2=0x0002 Assume value at 0x0002 in memory = 0x0006 check that $r1=0x0005	xor $r1, $r2, 100 1001000100100100 if the flag does not match CC, make sure the value in $r1 is unchanged, otherwise do previous. Use initial conditions from first column	
or	1100	or $r1, $r2 1100000100100111 input: $r1 = 0x0003, $r2= 0x0005 check that $r1 = 0x0007 after execution	or $r1, ($r2) 1100000100101111 input: $r1 = 0x0003, $r2=0x0002 Assume value at 0x0002 in memory = 0x0006 check that $r1=0x0007	or $r1, $r2, 100 1100000100100100 if the flag does not match CC, make sure the value in $r1 is unchanged, otherwise do previous. Use initial conditions from first column	
push	1010	push $r1 1010000100000111 Value of stack register is 0x0004 initially. Check whether the value in the register is now 0x0006. if $r1 is initially 4 Check if Mem[stack register] = 4		push $r1, 100 1010000100000100 if the flag does not match CC, make sure stack pointer register is not incremented, otherwise do previous. Use intitial conditions from first column	
pop	1011	pop $r2 1011001000000111 Value of stack register is 0x0004 initially. Check		pop $r1, 100 1011000100000100 if the flag does not match CC, make sure stack pointer	

		whether the value in the register is now 0x0002. if Mem[stack register] = 4 initially, check if $r1=4.		register is not decremented, otherwise do previous. Use intitial conditions from first column	
jr	1110	jr $r1 1110000100000111 check the PC if it is the same value as $r1		jr $ra, 010 1110111000000010 if the flag does not match CC, then PC should be PC+1	
jal	1111	jal label 1111111000001111 (label's address) if this is instruction 0x0202 and the label is		jal label, 010 1111111000001010 (label's address) if the flag does not match CC, then PC should be PC+1	jal label 1111111000001111 (invalid address) The label doesn't exist. Should cause an error.
jl		jl label 1111000000000111 (label's address) check the PC to make sure the branch was executed, PC = PC + 2		jl label, 100 1111000000000100 (label's address) if the cc is true, make sure the branch was executed. make sure the branch was executed.	jl label 1111111000001111 (invalid address) The label doesn't exist. Should cause an error.

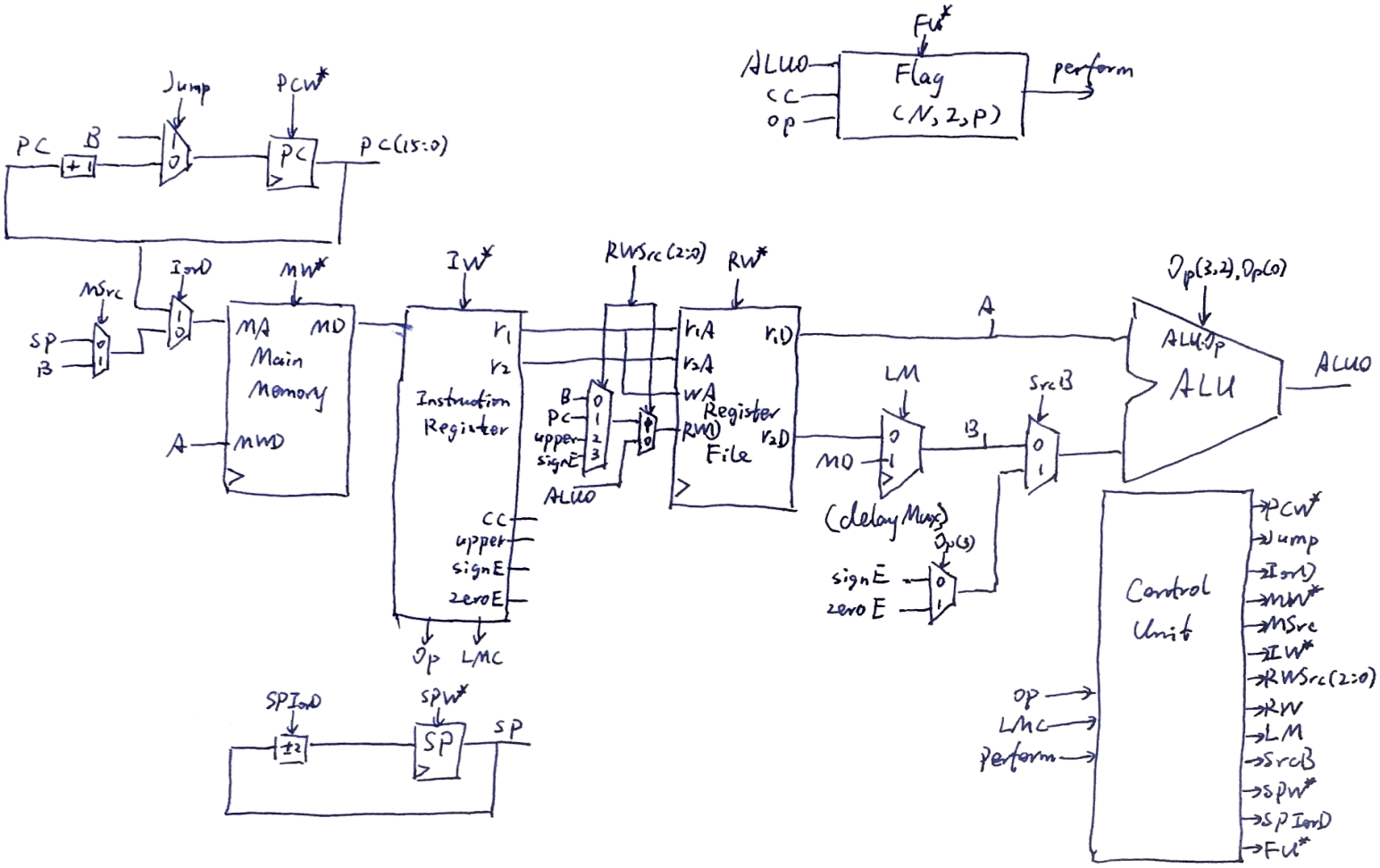
Connections of the Datapath Components:

Table 7 Connections of Datapath Components

Type of component	Name	Inputs					Outputs			
		0	1	2	3	**Control Signal**	0	1	2	**Control Input**
Mux	MSrcMux	SP	B			**MSrc**	MDAddr			
	PCMux	PC+2	B			**Jump**	newPC			
	RWSrcMux	ALUO	SpeRWD			**RWSrc(0)**	RWD			
	SrcBMux	B	SrcBimm			**SrcB**	ALUB			
	SrBimmMux	signE	zeroE			Op(3)	SrcBimm			
	LMDelayMux	r2D	MD			**LM**	B			
	IorDMux	MDAddr	PC			**IorD**	MA			
	RWSrcMux2	B	PC	upper	signE	**RWSrc(1:0)**	SpeRWD			
Memory/ Register	Memory	MA	MWD			**MW**	MD			
	Register File	r1A	r2B	RWD		**RW**	r1D	r2D		
	PC	newPC				**PCW**	PC			
	Instruction register	MD				**IW**	r1	r2	CC	**Op**
							upper	signE	zeroE	**LMC**
	Stack Pointer	newSP				**SPW**	SP			
	Flag	ALUO	CC	Op		**FU**				**perform**
ALU/ Adder	ALU	A	ALUB			**ALUOp**	ALUO			
	SPAdder	SP				**SPIorD**	newSP			
	PCAdder	PC					PC+2			

Datapath Design Diagram

All signals with arrows into block are control signals, while all arrows out of blocks are input to control.



Description of Datapath Components

Note for text in *Italic*: RESET control to initialize our datapath, and modification necessary due to Xilinx's limitation of 14-bit address is described above, but not yet implemented yet. We will work on those updates in next milestone.

Table 8 Datapath components Desriptions

Type of component	Name	Description
Mux	MSrcMux	Memory source mux indicates whether pop or branch is used, then select the correct memory location then send it to the main memory. It takes three inputs: SP, the stack pointer; B, the output from the register file; and the MSrc bit from the control. This MUX outputs the address of memory read or write.
	PCMux	The jump mux is used to determine the address of next instruction, it could be PC+1, or the jumping address in the B register. This mux is controlled by jump signal, it uses the PC+1 by default, when jump is called, it is set to 1, and set PC to the address that is stored in the B register
	SrcBMux	SrBMux controls the second input of the ALU, it could be either from the sign extension when Lui is used, or from register B when R type, J type instructions is called. This mux is controlled by SrcB signal, when using R type instruction, it will send out the result from B register to the ALU, else it will take the.
	LMDelayMux	LM determine the value of B, if R type instruction is used, it will take the register regFile(3-6) value, else it will use the value from memory data. It needs three inputs: r2D, which is the data from the second register; MD, which is data taken from memory; and the LM bit from the control. LM, the control signal, will be delayed for one cycle. If LM = 1, then in the next cycle, B = MD, otherwise B = r2D.
	IorDMux	Instruction or data determine if the datapath is moving on to the next instruction in memory, jump to a specific instruction or doing push/pop to alter the stack in the memory. The mux takes three inputs: PC, which is is the value of the program counter, or the value from MSrc. This MUX outputs either the PC, so that the Main Memory can retrieve that instruction; sp, so that main memory can get that information off of the stack; or B, which would be if an R-type instruction accessed memory to get the value.
	RWSrcMux2	The regWrite source mux is used when register write needs value besides ALUO. Only a few instruction would use it-- If cp is used, it will select B(0); If lui is used, then it will select the value from upper(2); if cpi is used, it will select immediate from signE(3); if jal is used, it will select PC(1) to write into $ra. The output of the result will be used to prepare to write into the register file.

	RWSrcMux	This mux is used when the instruction need to write in a register, it will use ALUO when arithmetic operation needed to be performed in the instruction, otherwise it will use the result from RWSrcMux2.
Memory/ Register	Main Memory	Memory stores instruction data, each instruction is 2 bytes, and it can only be accessed by chunk of 16 bits. It takes four inputs: MRA, which reads the specified address; MWA, which chooses a specified memory location to write to; MWD writes the specified data to the address specified in MWA; and the MW control bit. The output of this memory block is MD, memory data, which is the data from the memory address specified in MRA. *In actually implementation, Xilinx can only generate a 14-bit memory, so we decide to throw the 11 and 10 bit of our memory. Memory should throw an exception if MA[11]|MA[10] is not 0. Also, to protect procedure stack from malicious access, memory would also throw an segment fault if MA is bigger than $sp.*
	Register File	The Register File can read and write data to 16 bits registers that used in assembly, each register is represented by a 4 bit number. The file takes four inputs: r1A, which is the address of $r1; r2A, which is the address of $r2, RWD, which is the value to write to the address specified in r1A; and the RW control whether to write or not. The output of this register is r1D, which is the data at r1A, and r2D, which is the data value at r2A. R1D and r2D would output data as soon as address appear at r1A or r2D, but it only write data at rising edge of clock when RW = 1. The newly written data should appeared at r1D immediately after write.
	PC	This is a register that store the current location of current instruction in the main memory. It can be updated through jump with value from B.
	Instruction register	The instruction register reads the data coming out of the memory, and spits out a single instruction. The CC is used in R, J, and J type instructions. They are specified in the assembly code. r1(8-11) and r2(4-7) are both outputted as the register's addresses specified in the instruction. It outputs LMC and Op to control. It also outputs upper, the immediate used for lui, and signE, the immediate used for cpi and addi.
	Stack Pointer	It takes the control bit SPW (stack pointer write) so the user can push and pop. It also take the input of an updated stack pointer that was either incremented or decremented by 1. It outputs the stack pointe.

		Since no programmer is able to access it, it would have a RESET control, to initialize STACK Pointer to 0xf3ff – the top address of our stack.
	Flag	This flag register takes the input from ALU output, CC code, and Op code. The flag determines whether this instruction needs more than 2 cycle. It has one output Perform to control unit, which will determine whether control unit would cut this instruction and start fetch next instruction in the following cycle. For non I-type instruction, at cycle 2, flag compares CC and flag bit and output Perform = 1 if they match. The rest of instruction is skipped when they don't match and Perform = 0. (See condition code specification in previous section for when CC matches flag). So a skipped instruction only takes 2 cycles. For I-type instruction, flag cannot perform CC since I-type instruction does not even have CC bit, I-type instruction is always executed. In this case, Perform would be 1 if this I-type instruction would needs more than two cycle, eg. addi; Perform would be 0 if this I-type instruction is finished after cycle 2, eg cpi, lui. In other word, when Op = 0001(addi), Perform is always 1, and when Op = 0111(cpi) or 0011(lui), Perform is always 0. Flag bit would be updated when FU is 1 according to output of ALU result. It would store the sign of ALU result in 3 bits—N(negative), Z(zero), P(positive). Note that Flag does not get take input from ALUO, but the directly output of ALU (ALUOut in the component table), so updating can be perform in the same cycle of ALU calculation.
ALU/Adder	ALU	ALU reads inputs from SrcA and SrcB, then computes the result according to ALUOp (the ALU operation code). The output of the ALU will be written into register file immediately in the same cycle of calculation.
	SPAdder	This adder increments or decrements the value of SP by 1 depending on the control signal from SPIorD. Since Xilinx implementation can only contain 14-bit address, SPAdder would ignore SP[11] and SP[10] in addition and always output 0 for them.
	PCAdder	This is an adder that increments the PC by 1 and send the result to the PCMux. Since Xilinx implementation can only contain 14-bit address, PCAdder would ignore SP[11] and SP[10] in addition and always output 0 for them.

Testing of Each Component

Table 9 Component testing

Component	test
16bits_2to1 mux	Set the following: A = 0xa; B = 0xc; op = 0; output = 0xa A = 0xa; B = 0xc; op = 1; output = 0xc if the output of the mux is correct in both cases, then the mux is working correctly.
16bits_4to1 mux	Assume A,B,C,D, and op are inputs into the mux and op is the control. Set the following: A=0xa;B=0x2;C=0xb;D=0xc;op=11; output = 0xc A=0xa;B=0x2;C=0xb;D=0xc;op=10; output = 0xb A=0xa;B=0x2;C=0xb;D=0xc;op=01; output = 0x2 A=0xa;B=0x2;C=0xb;D=0xc;op=00; output = 0xa If the outputs are all correct, then the mux works correctly.
Main memory	Memory read: assume the memory has only one instruction: Instruction 0x0000 Input: IorD=1, MW = 0 output:0x0
	Memory write: Input: MW=1, A = 0xffff result: mem[A] =0xffff
Register File	Input: RA1 = 4'b0010, RWD = 16'h1425, RW = 1 wait after one rising clock edge Input RA1 = 4'b0100, RWD = 16'h77ED, RW = 1 wait after one rising clock edge

	Input: r1A = 0010 , r2A = 1000, $t2 = 0x1425, $s0 = 0x77ed check if r1D = 1425, r2D =0x77ed if it does, then register write and read both works
ALU	Assume A, B, and op (control) are the inputs into the ALU for all the ALU tests Add tests: Set the following: (P + P = P) A=0xa; B=0xb; op = (000) Output: ALU0 = 0x15 Set the following: (N + P = P) A=0xfff6; B=0xb; op = (000) Output: ALU0 = 0x0001 Set the following: (N + P = N) A=0xa; B=0xfff5; op = (000) Output: ALU0 = 0xffff Set the following: (N + N = N) A=0xfff6; B=0xfff5; op = (000) Output: ALU0 = 0xffeb Set the following: P + P = overflow A=0x0001; B=0x7fff; op = (000) Output: ALU0 =0x8000 Overflow = 1 Set the following: N + N = overflow A=0x8000; B=0x8000; op = (000) Output: ALU0 = 0x0 Overflow = 1 Set the following: A=0x0; B=0x0; op = (000) Output: ALU0 = 0x0 If all 7 cases product the correct output, ALU add works correctly.
	ALU tests Subtraction tests: Set the following: (P - P = P) A=0xb; B=0xa; op = (010) Output: ALU0 = 0x1 Set the following: (P - P = 0)

	A=0xb; B=0xb; op = (010) Output: ALU0 = 0x0 Set the following: (N - P = N) A=0xfff5; B=0xb; op = (010) Output: ALU0 = 0xffeb Set the following: (N - N = N) A=0xfff5; B=0xfff6; op = (010) Output: ALU0 = 0xffff Set the following: P - P = N A=0xa; B=0xb; op = (010) Output: ALU0 = 0xffff Set the following: N - P = overflow A=0x7fff; B=0xffff; op = (010) Output: ALU0 = 0x8000 Overflow = 1 Set the following: N - N = P A=0xfff6; B=0xfff5; op = (010) Output: ALU0 = 1 If all 7 cases product the correct output, ALU sub works correctly.
	use the following inputs to check to make sure or's correct outputs are produced: input: A=(1001 0010 0000 0000) B =(0000 0111 1001 1111); op = (110) output: ALUO = (1001 0111 1001 1111) input: A= 0xffff B = 0xffff; op = (110) output: ALUO = 0xffff input: A= 0 B = 0; op = (110) output: ALUO = 0 input: A = 0 B= 0xffff; op = (110) output: ALUO= 0xffff if all 4 test cases produce the correct output, the ALU works correctly for or.
	use the following inputs to check to make sure xor's correct

	outputs are produced: input: A=(1001 0010 0000 0000) B =(0000 0111 1001 1111); op = (101) output: ALUO = (1001 0101 1001 1111) input: A= 0xffff B = 0xffff; op = (101) output: ALUO = 0 input: A= 0 B = 0; op = (101) output: ALUO = 0 input: A = 0 B=0xffff; op = (101) output: ALUO= 0xffff if all 4 test cases produce the correct output, the ALU works correctly for xor.
	use the following inputs to check to make sure and's correct outputs are produced: input: A=(1001 0010 0000 0000) B =(0000 0111 1001 1111); op = (100) output: ALUO = (0000 0010 0000 0000) input: A= 0xffff B = 0xffff; op = (100) output: ALUO = 0xffff input: A= 0 B = 0; op = (100) output: ALUO = 0 input: A = 0 B= 0xffff; op = (100) output: ALUO= 0 if all 4 test cases produce the correct output, the ALU works correctly for and.
Instruction Register (r1, r2, CC)	Input instruction add $t0, $t4; 0000 0000 0100 0111; check if op is 0000, r1 is 0000, r2 is 0100, CC is 111.
	Input first half of jl label , 010; 1111 1110 0000 1010; check if op is 1111, CC is 010.
(signE)	Input instruction addi $t3, -2; 0001 0011 1111 1101; check if op is 0001, r1 is 0011, signE is 0xfffe
(upper)	Input instruction lui $s2, 5; 0011 1100 0000 0101; check if op is 0011, r1 is 1100, upper is 0x0500,
Flag	input: CC = 101, op = 0000, Flag bits [N,Z,P] = 100, FU = 0

	check if output (perform) is 1 input: CC = 111, op = 0100, Flag bits [N,Z,P] = 010, FU = 0 check if output is 1 input: CC = 001, op = 1100, Flag bits [N,Z,P] = 010, FU = 0 check if output is 0 input: CC = 001, op = 0001, Flag bits [N,Z,P] = 010, FU = 0 check if output is 0 (immediate type always output 0)
Flag(update)	input: ALUO = 0x ffff, FU = 1 check if [N,Z,P] = 100 after a rising clock cycle. input: ALUO = 0x 0020, FU = 1 check if [N,Z,P] = 001 after a rising clock cycle. input: ALUO = 0x 0000, FU = 1 check if [N,Z,P] = 010 after a rising clock cycle.
PC adder	input: PC = 0x0006 check if output is 0x0007 input: PC = 0xffff check if output is 0x0000 input: PC = 0xfffe check if output is 0xffff
SP adder	input: SP = 0x0006 IoD=1 check if output is 0x0007 input: SP = 0x0000 IoD=0 check if output is 0xffff input: SP = 0x0001 IoD=0 check if output is 0x0000 input: SP = 0xffff IoD=1 check if output is 0x0000

Integration Tests

Note: “x” symbolizes any value. All hex values are preceded by “0x”. Everything else is in binary.

Set of Components	Inputs	Outputs	Tests
PC + Memory + IR	PC, IW	R1, R2, CC, LMC, Op, Upper, signE	Initial state: Mem[PC] = 0x4111, IW = 1 Expected Output: R1=0x1, R2=0x1, CC=001, LMC=0, Op=0x4, Upper=0x1100, signE=0x0011

			Initial state: Mem[PC] = 0x8eb8, IW = 1 Expected Output: R1=1110, R2=1011, CC=000, LMC=1, Op=1000, Upper=0xb800, signE=0xffb8
SP + PC + Memory	PC, SP, B, MSrc, MW, A	MD	Input: Mem[0x0001] = 0x0022, MSrc=0, IorD=0, SP=0x0001, MW=0, PC = 0x0000, A=0x0002 Expected Output: MD=0x0022 Input: Mem[0x00F3] = 0x0744, SP=0x00F3, MSrc=0, IorD=0, MW=1, A=0x0044 Expected Output: MD=0x0744 first, After clock edge MD=0x0044 Input: B=0x00F3, A=0xFF00, MW=1, IorD=1, MSrc = 1; Output: Check memory at 0x00F3 is 0xFF00
RF + ALU	R1A, R2B, RWD, RW, ALUOp, LM, SrcB, MD	ALUO, Overflow	Assume $0 holds 0x0004 Assume $1 holds 0x0002 Input: R1A=0x0, R2B=0x1, RWD=0000, RW=1, ALUOp=010, LM=0, SrcB=0, MD=x Expected Output: ALUO=0x0002, Overflow=0 Assume $0 holds 0x0000 Assume $1 holds 0x0000 Input: R1A=0x0, R2B=0x1, RWD=0000, RW=1, ALUOp=000, LM=1, SrcB=0, MD=0x0000 Expected Output: ALUO=0x0000, Overflow=0 Assume $0 holds 0x0004 Assume $1 holds 0x0004 Input: R1A=0x0, R2B=0x1, RWD=0000, RW=1, ALUOp=000,

			LM=0, SrcB=1, MD=x Expected Output: ALUO=0x0008, Overflow=0 Assume $0 holds 0xFFFE Assume $1 holds 0x0002 Input: R1A=0x0, R2B=0x1, RWD=0x0, RW=1, ALUOp=000, LM=0, SrcB=0, MD=x Expected Output: ALUO=0x0001, Overflow=1
IR + Flag	MD, ALUout, FU*	LMC, Upper, Perform	Input: MD=0x1001, ALUout=0x0002, FU*=0 Expected Output: LMC=0, Upper=0x02, Perform=1 Input: MD=0x0010, ALUout=0x0000, FU*=1 Expected Output: LMC=0, Upper=0x10, Perform=1 Input: MD=0x04015, ALUout=0x0000, FU*=0 Expected Output: LMC=0, Upper=0x15, Perform=0
IR + RF	MD, B, PC, RWSrc, ALUO	R1D, R2D, Op, LMC	Assume $0 holds 0x0000 Assume $1 holds 0x0000 Input: MD=0x0007, B=x, PC=x, RWSrc=000, ALUO=0x0000 Expected Output: R1D=0x0001, R2D=0x0001, Op=111, LMC=0 Assume $0 holds 0x0000 Assume Mem[$1] holds 0x0008 Input: MD=0x001F, B=0x421, PC=x, RWSrc=000, ALUO=0x0008 Expected Output: R1D=0x0000, R2D=0x0008, Op=111, LMC=1 Assume $0 holds 0x0002 Input: MD=0xE017, B=0x0000,

			PC=x, RWSrc=000, ALUO=x Expected Output: R1D=0x0002, R2D=x, Op=1110, LMC=0 Assume $0 holds 0x0000 Assume $1 holds 0x0000 Input: MD=0xD017, B=0x0000, PC=x, RWSrc=000, ALUO=0 Expected Output: R1D=0x0001, R2D=0x0002, Op=1101, LMC=0
ALU + Flag	A, B, CC, Op, FU, ALUOp, Flag*	ALUO, Perform	Input: A=0x0008, B=0x0001, CC=011, ALUOp=010, Op=0100, FU=0, Flag*=001 Expected Output: ALUO=0x0007, Perform=1 Input: A=0x0001, B=0x0001, CC=010, ALUOp=000, Op=0010, FU=0, Flag*=001 Expected Output: ALUO=0x0002, Perform=0 Input: A=0x0000, B=0x0008, CC=111, ALUOp=010, Op=0101, FU=1, Flag*=100 Expected Output: ALUO=0xFFF9, Perform=1
PC+adder+mux	PC, B,Jump, PW*	PC	Input:PC = 0, B = 12, Jump = 1, PW*=1 Expected Output: PC = 12 Input: PC = 0, B = 12, Jump = 0, PW*=1 Expected Output: PC = 1
SP+ addder	SPW, SPIorD, SP	SP	Input: SPW = 1, SPIorD = 1, SP = 10 Expected output: SP = 11 Input = SPW = 1, SPlorD = 0, SP = 11

			Expected output: SP = 10
pc+adder+mem+IR+sp	SPW, SPIorD, SP, MW,PCW, IorD, Write, MA, MWD	PC, Op, signE, SP, r1	input: 0111 1001 0001 0001 expected : Sp = 0xff3f, Op = 0111, r1 = 1001, signE = 10001 input: 1010 1001 0000 0111 expected : Sp = 0xff3e, Op =1010, r1 = 1001 input: 1011 1100 0000 0111 expected: Sp = 0xff3f, Op= 1011, r1 = 1001
mem+RF+alu+flag	KK		

Ideas for Xilinx implementation

Most of our components are implemented with Verilog, a powerful hardware description language.

Muxes can be easily implemented with case switch statements or ? : in Verilog.

Memory is pretty complex and slow. We will use a block memory to save more resources for other component. However, block memory is slower. We designed our RTL assuming that we have an asynchronous memory unit in the first time. By the time we run into issue of synchronous block memory, our RTL and control design is already fixed. We do not really want to change our RTL and control unit, or make memory read two cycle.

First option we have is to Instruction register is just a 16-bit register with many different outputs. Each output is a part of instruction. We can use { , } syntax in Verilog to create new buses. For example, signE[15:0] is { inst[11] , inst[11], inst[11], ….. , instrct[11:0] }. All outputs such as r1, r2, CC, Upper and sign Extended are certain bits of the instruction are only connected to inst with wires, so there shouldn't be any delay in instruction register. When IW is high, Instruction register would prepare itself for the upcoming instruction from memory and update itself in the next cycle when memory read is completed. See Memory implementation section above more details.

Register file is just a group of 16 16-bit registers. To implement this with Verilog, we can directly declare an array of 16 register, and assign r1D to Reg[r1], r2D to Reg[r2].

ALU will be implemented with Verilog with cases. It takes advantage of Verilog's ability to do multi-bits design. With "case switch", we can describe the output of ALU given specific input, and Verilog magic comes into play and generate corresponded hardware. We can do an

overflow check after we have the results. There are still a few means that we can improve our ALU right now. See Design Journal for more details.

Memory is a bit tricky. To give memory a whole cycle to operate and reduce our clock cycle, we decided to make memory synchronous and make modifications of instruction register and B. Details about this modification is in the design document.

Flag is implemented with Verilog. It has three flip flops, combinational logic to determine Perform. Perform is asynchronous not controlled by clock.

Control signals

We have 14 control signals and 3 inputs into the datapath.

Note PCW, MW, IW, RW, SPW, FU have a * in the table. This * specifies that those signals control register write. These write signals always matter! If it is not specified, the control signal is 0. If a control signal not listed above is not specified, it means its value does not matter.

The “when it matters” column specifies when the control signal is important. Only when the condition in this column is met do we care about this control signal. These “don’t care” cases introduce more design flexibility for the control unit.

Table 10 Control signals

Control signals	Description	When it matters	on (= 1)	off (= 0)
PCW* (PC Write)	Controls if the PC register can be overwritten with a new value.	Always	PC is updated	PC is unchanged
Jump	Controls whether PC’s new value will be the next instruction or the address from a jump.	PCW = 1	PC jumps to a new address stored in B right now	PC will increment 2.
MW* (Memory Write)	Controls whether the datapath can write to the memory block during a store.	Always	Memory will write data.	
IorD (Instruction or Data)	Controls whether or not the next line read from	IW = 1 or LM = 1	Memory reads an instruction from	Memory reads or writes data

	the Instruction Register should be treated as an address or instruction.	or MW = 1	Mem[PC] (MW and IorD should never be on at the same time)	at address specified by MSrc.
MSrc (Memory address source)	Controls which address memory is reading or writing data.	IorD = 0 and (IW = 1 or LM = 1 or MW = 1)	Memory reads or write data at address of B	Memory reads or write data at address of SP.
IW* (Instruction Write)	Controls whether or not the Instruction Register can be overwritten by loading from memory.	Always	Instruction register loads memory from memory	Instruction register remains unchanged. (when IorD = 0, IW should never be 1)
RW* (Register Write)	Controls whether or not the register block can be written to.	Always	Write into register file	Register file remains unchanged
LM (Load Memory)	Controls where B will contain the data from memory or data from register file.	When we need to use B next cycle	B will have data at address specified by MSrc in the next cycle. (IorD should always be 0 when LM = 1)	B will have data of Reg[r2] in the next cycle.
RWSrc(0) (Register Write Source)	Controls if data is written from a special source.	RW = 1	Write data into register file from some special source from B, PC, signE, upper	Write ALUO into register file at address r1. R[r1] = ALUO
RWSrc(2:1) (Register Write Source)	Controls what data should be written into the register file.	RW = 1 and RWSrc(0) = 1	Select data to write into register file as 00 -- B ; 01 -- PC ; 10 -- upper ; 11 -- signE	
SrcB (Source B)	Controls which input should be the B input into the ALU.	When we need ALUO next cycle	use B as the second source of ALU, otherwise	use sign-extended immediate as the second source of ALU
ALUOp(2:0) (ALU Operation)	Controls which operation the ALU does on A and B.		ALU does following operation on A and B 00x -- add ; 01x -- subtract ; 100 -- and ; 101 -- xor ;	

			110 -- or ; (We don't know what ALU will do if ALUOp is not included above.) (We might decide to add more operations later)	
SPW* (Stack Pointer Write)	Controls whether or not the stack pointer should be updated.	Always	write and update the stack pointer depending on SPIorD	stack pointer remains the same
SPIorD (Stack Pointer Increment or Decrement)	Controls whether to increment or decrement the stack pointer based on the instruction.	SPW = 1	increment SP by 2	decrement SP by 2
FU* (Flag Update)	Controls whether or not the control code should be updated based on the value of ALUO.	Always	Update flag bits based on sign of data in ALUO	ALUO remains the same

Control signals for RTL for each possible cycle

As explained above, there might be an extra cycle between cycle 2 and cycle 3 for loading memory (box in red) for certain instructions if LM = 1.
The following two tables show the control signals for each cycles in RTL.

For control signals that are not specified in a particular cycle, it is only important if it is set to zero.

Table 11 Control signals of RTL first half

	Calculations	cmp	sto	cp	jr	jl	jal
Cycle 1	IorD = 1 PCW Jump = 0						
Cycle 2	IW LM = 0					IW PCW LM = 1 IorD = 1	
LM cycle	LM = 1						

Cycle 3	SrcB = 0 ALUOp = op RW RWSrc = xx0	SrcB = 0 ALUOp = (A - B) FU	MW IorD = 0 MSrc = 1	RW RWSrc = 001	PCW Jump = 1	PCW Jump = 1 RW RWSrc = 011

Table 12 Control signals for RTL second half

	addi	ori	lui	cpi	push	pop
Cycle 1	IorD = 1 PCW					
Cycle 2	IW	IW	IW RW RWSrc = 101	IW RW RWSrc = 111	IW SPW SPIorD = 0	IW LM = 1 IorD = 0 MSrc = 0
Cycle 3	SrcB = 1 ALUOp = (A + B) RW RWSrc = xx0	SrcB = 1 ALUOp = (A | B) RW RWSrc = xx0			MW IorD = 0 MSrc = 0	SPW SPIorD = 1 RW RWSrc = 001

State transition diagram

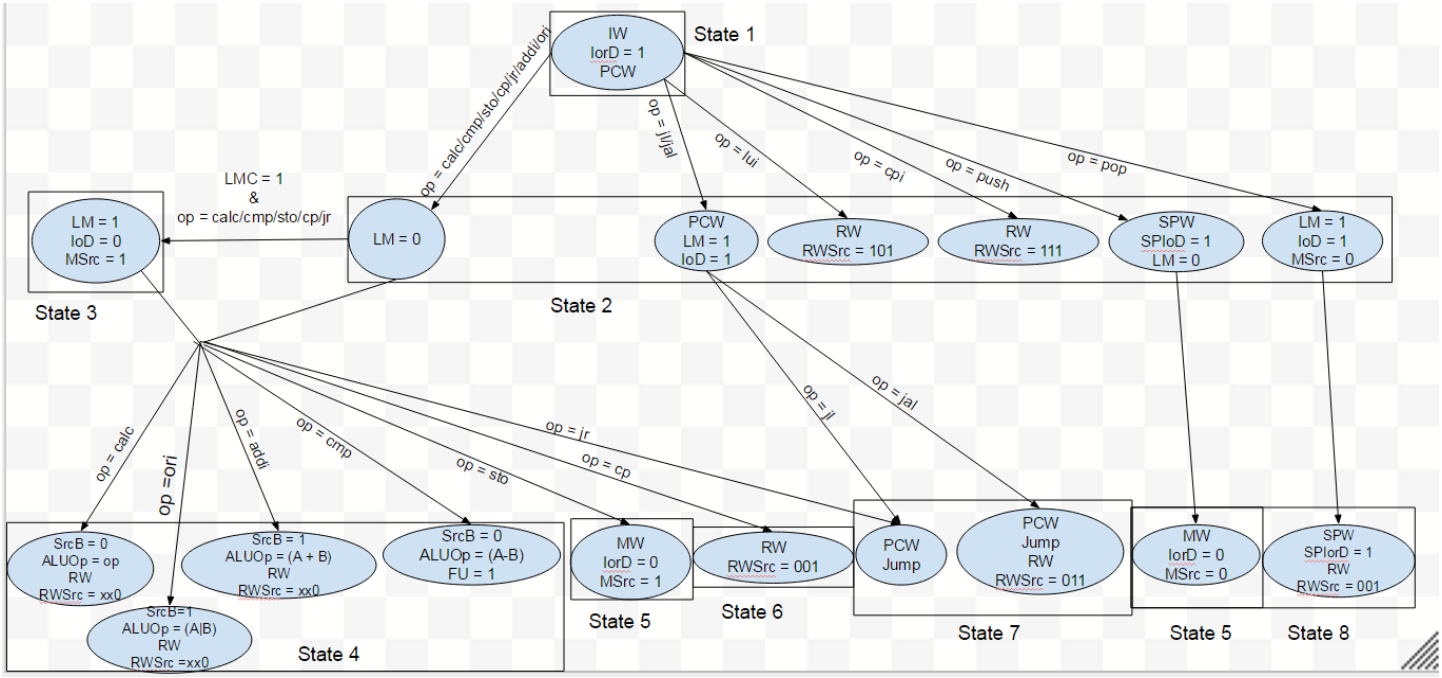
Table 13 State Transition

State	Perform	op				instruct	Next State	
		op(0)	op(1)	op(2)	op(3)		LMC = 0	LMC = 1
State 1: Read instruction	x	x	x	x	x	any inst	2	-
State 2: Decode / load register value / write immediate	0	x	x	x	x	any inst	1	1
	1	0	0	x	x	calc inst	4	3
	1	1	0	0	0	addi	4	4
	1	1	1	0	1	ori	4	4
	1	1	0	1	0	cmp	4	3
	1	1	0	0	1	xor	4	3
	1	0	1	0	0	sto	5	3
	1	0	1	1	0	cp	6	3
	1	0	1	0	1	push	5	6

	1	0	1	1	1	jr	7	3
	1	1	1	0	0	lui	1	1
	1	1	1	1	0	cpi	1	1
	1	1	1	0	1	pop	8	8
	1	1	1	1	1	jl/jal	7	7
State 3: Load memory data	1	x	0	x	x	calc inst	4	-
	1	0	1	0	0	sto	5	-
	1	0	1	1	0	cp	6	-
	1	0	1	1	1	jr	7	-
State 4: Calculations	1	x	x	x	x	any inst	1	-
State 5: memory write	1	0	1	0	0	sto	1	-
	1	0	1	0	1	push	1	-
State 6: cp	1	0	1	1	0	cp	1	-
State 7: jump	1	0	1	1	1	jr	1	-
	1	1	1	1	1	jl	1	-
	1	1	1	1	1	jal	1	-
State 8: pop	1	1	1	0	1	pop	1	-

State “bubbles”

Table 14 State Bubbles for each instruction



	Calc	cmp	sto	cp	jr	jl	jal	addi/ori	lui	cpi	push	pop
Cycle 1	State 1: Read instruction											
Cycle 2	State 2: Decode / load register value / write immediate											
LM cycle	State 3: Load memory data											
Cycle 3	State 4: Calculations		State 5: memory write	State 6: cp	State 7: jump			State 4: Calculations			State 5: memory write	State 8: pop

Figure: The above chart shows the state for each instruction and its cycles. State 3 is optional and only used in R type instructions -- if the LM bit is set to true, then State 3 is used, otherwise it is skipped. For example, calculation with load memory has state 1, 2, 3, 4, and 9, calculation without load memory will have states 1,2,4 and 9. For other instructions like Jal, it will have states 1, 2 and 7.

State	Instruction	OP	Output
1 Fetch Instruction	all inst	0000	PCW=1, Jump=0, IorD=1
2 Decode	other inst	0000	LM = 0, IW=1
	addi	0001	IW = 1
	ori	1101	IW = 1
	push	1010	SPW = 1, SPIorD = 0, IW=1
	pop	1011	LM=1, IorD=0, MSrc= 0 , IW=1
	jl/jal	1111	PCW=1, Jump=0, LM=1, IorD=1, IW=1
	lui	0011	RW = 1, RWSrc = 101, IW=1
	cpi	0111	RW = 1, RWSrc = 111, IW=1
3 Load memory	all inst	xxxx	LM = 1, IorD = 0, MSrc = 1
4 Calculation	calc op	xxxx	SrcB = 0 ALUOp = op, RW = 1, RWSrc = xx0
	Andi	0001	SrcB = 1 ALUOp = 000, RW = 1, RWSrc = xx0
	ori	1101	SrcB = 1 ALUOp = 111, RW = 1, RWSrc = xx0
	cmp	0101	SrcB = 1 ALUOp = 000 FU
5 Memory write	sto	0010	MW= 1, IorD = 0, MSrc = 1
	push	1010	MW= 1, IorD = 0, MSrc = 0

6 cp	cp	0110	RW=0, RWSrc=001
7 jump	jr	1110	Jump = 1, PCW = 1
	jl	1111	Jump = 1, PCW = 1
	jal		Jump = 1, PCW = 1, RW= 1, RWSrc= 011
8 pop	pop	1011	RW=1, RWSrc = 001, SPW=1, SPIorD = 1

Control Signal output

The figure above shows if the control works correctly when the op code is given. The first column shows the state of the current process; the second column shows the instruction is running; and the third column is the operation code. LMC is only used in states 2 and 3, and it indicates if memory access is needed at this stage. Perform is used to decide whether the instruction will be executed or not. The output column shows the expected control signal generated from the state.

Control Unit Xilinx implementation design

The Control Unit will be a one-hot mealy state machine. As long as we have the state diagram we can directly implement it. Our Control Unit will be a mealy machine, so our output depends not only on the state but also the input. Each instruction starts to do different work at cycle 2. The number of cycles can be reduced further. In our project, we implement two control units, one with behavior level Verilog, which is easier to implement, and another with register-transfer level Verilog, which takes full use of “don’t cares” states to maximize performance. The latter is harder, because we would need to draw K-Maps and find the optimal gate design manually. The former is easier and serves as a backup plan. See the Design Journal for more details about control unit implementation and testing.

Control Unit Testing

Control unit testing will involve running all combinations of Op, LMC and Perform in the state diagram and checking its control signal output and state transition in the above table. All cases will be tested exclusively. However, it will be tedious to test every case, so we created another module called ControlUnitTester to help us.

ControlUnitTester module takes state, input to controls, and outputs corresponding control signals and the next state. For control signals that don’t matter, ControlUnitTester will output an extra bit to specify whether this signal is necessary. ControlUnitTester is implemented simply by copying each in FSM.

ControlUnitTester computes answers rather than storing them. **In the test bench, we can run ControlUnit with a specified input combination for multiple cycles and check the output and transition to ControlUnitTester.** The checking also covers the case of where the control signal doesn't matter. In actual code, for example, MSrc is the output from ControlUnit, and is compared with the standard answer MSrca from ControlUnitTester. If they equal or MSrca indicates "don't care", then MSrc is 1 to indicate that MSrc passes the test in this case. If any control signal does not pass the test, "error" will be high. Sometimes, our FSM will transmit to some impossible state, with all control signals set to "don't care". We do not want to miss those errors, so ControlUnitTester will also output an error signal.

System Test

To test our full datapath, we will use instructions that have been previously specified in this document. To test individual instructions, we will use the tests described in the RTL testing methods; to test smaller groups of instructions, we will use the groups written in the Assembly Language Fragments for difficult instructions. Once all of those previous tests are passed, we test Euclid's algorithm. Because Xilinx takes time to load .coe files and these files need to initialize memory, we have decided to add muxes that allow for user input into our memory and register files to save time.

Also, to reduce repetitive code blocks in Verilog, we decided to use Verilog tasks to perform common operations like loading instructions into memory and specifying initial register file instructions. Five Verilog tasks are built in our system test bench.

run(PCstart , num) -- runs num instructions from PCstart.
readMem(address , data) -- read Mem[address] and store data for later comparasion.
writeMem(address , data) – write data into Mem[address]
readReg(address , data) -- read Rem[address] and store data for later comparasion.
writeReg(address , data) – write data into Rem[address]

With the help of those tasks, we can simply load the machine code into memory, call run to test them and check the value in memory and register. Writing Verilog is as easy as java!

Performance summary

1. The total number of bytes required to store both Euclid's algorithm and relPrime as well as any memory variables or constants.

Main function has 3 instructions, 10 Bytes, including one jl instruction at the end to create an infinite loop. RelPrime has 19 instruction, 42 Bytes. Gcd has 13 instructions, 30 Bytes. Altogether we have total 33 instructions, 82 Bytes program.

We only need 5 slots in the memory for backup register. The minimum memory run for E

euclid's algorithm is 82 + 5 * 2 = 92 Bytes.

2. The total number instructions executed when relPrime is called with 0x13B0 (the result should be0x000B using the algorithm specified in the project specifications).

   51090 instructions

3. The total number of cycles required to execute relPrime under the same conditions as Step 2.

   143029 cycles

4. The average cycles per instruction based on the data collected in Steps 2 and 3.

   2.7995 CPI

5. The cycle time for your design (from the Xilinx Synthesis report – look for the Timing summary).

   10.770ns

6. The total execution time for relPrime under the same conditions as Step 2.

   143029 cycles * 10.770ns/cycle = 1.5404ms

7. The gate count for your entire design (from the Xilinx Map report). This appears to have changed/is omitted in recent version. Extra credit for any group that finds a reasonable way to estimate the equivalent gate count from the data in the Xilinx reports.

   We can’t find relevant information

8. The device utilization summary

Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Flip Flops	21	9,312	1%	
Number of 4 input LUTs	71	9,312	1%	
Number of occupied Slices	36	4,656	1%	
Number of Slices containing only related logic	36	36	100%	

Number of Slices containing unrelated logic	0	36	0%	
Total Number of 4 input LUTs	71	9,312	1%	
Number of bonded IOBs	39	232	16%	
Number of BUFGMUXs	1	24	4%	
Average Fanout of Non-Clock Nets	3.20			

Design Journal

Runzhi Yang, Fred Zhang, Katrina Kerrick, Adam Fineral

Milestone 1

Choosing a combination architecture of load-store and accumulator

Load and store is going to make our processor run much faster because it needs less interaction with the memory. However, only 16 bits are available for one instruction, so there are not enough space for three register addresses if we want to have many registers. Therefore, we decided to have only two register addresses for the R-type and store the value back to the first one. This is an accumulator processor with multiple registers. It has an register to accumulate on and another register to specify another operand the address of the operand in the main memory.
This way, we would have one or two register address per instruction so we are able to have 16 different registers and more freedom and less dependence on memory.

Register types

We decided to have $ra and $sp so we can do function calls and return fast.

We decide to combine $t and $a because they use the same convention except when calling a function. They are both temporaries and destroyed after a function call. Therefore, we decided to just use the first n t-registers as the argument registers. So for a function with n arguments, $t0~$tn would store those arguments at the beginning of the function call. If there are more than eight arguments, they would have to be stored on the stack.

Zero register

We decided to leave out a zero register because the only use for a zero register is comparison. The way that our branch command works, we do not need a zero register, only the zero immediate.

Leave a 3-bit unused space in R-type

In our R-type, we have 4-bit opcode, two 4-bit register and 1-bit ML, so we have 3 bits left. We are not sure what to do with them for now. We might consider to add a function code for shift or something else of use. However, since these things are extra, we decided to focus on more important parts of the project for now.

B-type

When we are considering how to do optional branches, we have a hard time to putting everything into a 16-bit instruction. We decided to take the same idea from the IA32 processor from CSSE132 and to utilize some flag bits to control branches. So whenever the ALU performs some calculations, we would update the flag bits accordingly. Right now we have three flags -- the negative flag, the zero flag, and the positive flag. Only one flag of N, Z and P can be 1. Those flags would be used in the b instruction, which has a condition code (CC). b would check the flag from previous executed results and decide whether it needs to branch or not. Each bit in CC would correspond to one flag.

Generally speaking, for each compare and branch, we need to first subtract two values and store flags. Then in the second instruction, we can compare the flag value and the condition code in the instruction and perform the branch if necessary. For the purpose of ease of

programming, we would like to have 6 pseudo-instruction for branches -- beq, bne, blt, ble, bbl, bbe, and have our assembler to convert them into two real instructions

```
sub $ta,   $tb
b     CC   Label
```

CC stands for a 3-bit condition codes. From left to right, there is negative bit, zero bit, positive bit. There is a table available in Design document to indicate what CC is supposed to be for each pseudo-instruction.

Translation table from instruction to binary code

OPCODE	00	01	10	11
00	add	addi	sto	lui
01	sub	subi	cp	(blank)
10	and	andi	b	jal
11	or	ori	jr	j

Translation table from register name to register address

REGCODE	00	01	10	11
00	t0	t1	t2	t3
01	t4	t5	t6	t7
10	s0	s1	v0	v1
11	s2	at	ra	sp

*the operation codes go column then row

Explanation of tables

The instruction to binary table contains many instructions that are similar to MIPS instructions (i.e. add, addi, sub, and, j, etc). We would like to group instructions in the way above to simplify the ALU design. There are lots of instructions that ALU needs to perform and some of them are

very similar. We can actually reuse many blocks to make our processor smaller. For example, add, addi, sub, subi all use adders, so they are grouped together.
There are also a few unique ones. One such instruction is cp which was included to add the ability to copy one register to another register. Another added instruction is b. The b instruction does branches for a pseudo instruction. The last different instruction is subi, which is included currently, but may be removed later to allow room for an additional instruction. One operation code, 0111, does not refer to an instruction currently, so far we have two ideas for this register:

1. Make it cpi to copy and immediate value into a register
2. Make it a shift instruction, which will take a lot of thinking to create in hardware.

The other table contains the codes for all of the registers. We used mostly the same conventions as MIPS for their names. So, the eight t registers are all temporary, the three s registers are guaranteed to be the same across procedure calls, the two v registers are used to return values from procedure calls, the at register is used during sudo instructions, the ra register is used to store the return address across procedure calls, and the sp register stores the stack pointer.

Should we have ‘push’ and ‘pop’?

After writing the assembly code for the example program, we realized that we are using a lot of sto with addi 4 and cp with subi 4. According to the design principle that we should make common case faster, we would probably want to make these two cases faster. Using two instruction takes 5 + 4 = 9 cycles for cp and 3 + 4 = 7 cycles for sto, but if we combine them into one instruction, it is possible to do cp in 5 cycles and sto in 4 cycles. However, this change might include more work in datapath and control designing later. We feel that it would be something that we would probably like to do first after the basic design is settled.
The modification will have backward compatibility, so we don’t have to redo our previous work. For the instruction format, we can use those previous instruction name with only one argument. So r1 would remain the same, and the second data would be implies as popped out from the stack. We can use only one argument for sto, so by default, it simply pushes the register value onto the stack.
As for the binary encoding, we have 3-bit unused bit in R-type. Maybe we can make one of them a switch. So if the switch is on, we would simply perform pop or push with $bp and ignore 4 bits register address for r2.
We are not changing instruction specification for Milestone 1, but this would be something we might work on Milestone 2.

Ideas for exception handling

There is coprocessor registers in MIPS helping exception handling -- EPC, cause and status. EPC would stores the programmer counter where the exception happened. Cause stores the exception code and the flag bits when exception happened. Status stores the interrupt level, interrupt enable bit and any pending exception.
If we want to have our processor able to do exception handling, so sort of co processor like MIPS would have to exist. However, such co processor would require more instruction to move data in and out from the coprocessor, which would make our instruction set bigger.
To solve this problem without introducing new instructions, we considered several options:

1. Assign a few bytes in our memory to serve as ‘co processor’ and store EPC, cause and status when exception happened. So we have access to ‘coprocessor’ data with memory load and store instructions.
2. Still have a coprocessor, but forbid direct read and write with assembly instruction. When exception happens, EPC, cause and status are updated in the co processor, but a few of them are also put into t0, t1, etc as arguments of exception handlers. However, we need to backup those t registers into memory before load from coprocessor. The exception handler may update status, cause or other registers. Then after handler finishes its job, eret updates coprocessor registers with current t register values, restores original t0 and t1 from memory and jump to EPC.

Option 1 makes it easier to access and modify coprocessor register, but it’s pretty slow to have all coprocessor registers to memory.
Option 2 We need to do some extra data moving when exception happens and eret. Also, the more information we want our exception handler to know, the more restoration we need to do for t registers. Maybe we can stuff every information we need into one 16 bits register and only put one argument for exception handler.

Milestone 2

Instruction Set Changes

The instruction set saw many changes with this milestone. Many instructions were removed or changed into pseudo instructions. The only instruction that was removed entirely was subi. Subi wasn’t necessary as a negative sign in front of the immediate in addi would serve the same purpose. This is the same as the addi (big) pseudo instruction, and we determined that it would save time in the Assembler and was intuitive enough to not need a subi instruction at all.
If an instruction was changed to be a pseudo instruction, it was to make room for the new set of instructions.

We decided to include push and pop in the instruction set because they both lower cycle time and they are used very commonly in code. Push and pop, however, need to be a regular instructions to see this benefit.

The cmp command compares two register values and sets the comparison code. The purpose of the comparison code is explained more in full under the next heading.

Cp and cpi copy values and put them into a destination register. This is necessary because our add instruction only takes two register parameters instead of three, so copying a value must become a real command instead of a pseudo command.

Xor became a command because we thought it would be useful for arithmetic. It wouldn't be too hard to add to the ALU and it could come in handy, so we added it.

Jalr (jump and link register) is a command we will possibly be adding that functions exactly like the MIPS jal command except that it jumps to a register instead of a label. This is so that we have a way to go to functions while setting the return address register. It is currently unclear whether we will add this command into the group of j-type commands or if it will be standalone.

J, jal, and eret became pseudo instructions. The only real jump instruction now is jr. We decided that the J-type instruction should see a few changes to make room for what type of jump is being implemented. It now looks like this:

Opcode	register	jop	condition code

The opcode will indicate what type of instruction it is. Jop indicates what type of jump instruction it is (j, jal, eret, etc). Register indicates what register to look in for the jump address. 0 is an unused bit. Condition code is a three bit code that indicates when the jump should be made according to the comparison code.

Ori, andi, xori, clear, and addi (big) were added to the pseudo instruction set to make arithmetic easier. They were simple to add and cost us nothing.

Condition Code

Condition codes are very important to our instructions. The condition code is set when the instruction cmp is called.

```
cmp    $r1, $r2
```

The comparison code is three bits, with each bit representing a special value. Only one of the three bits will be 1 at a time! The other two will be zero. For example:

The first bit represents if $r1 < $r2. It is set to one if this is true, and produces a code of 100. If $r1 == $r2, then the code is 010. If $r1 > $r2, then the code is 001. While this could be done in two bits, doing it like this allows any command checking the condition code to check for not equals (101), greater than or equals (011), and less than or equals (110), which is three more values than if we had just used two bits.

The condition code is anded with the last three bits of each R-type, B-type, and J-type instruction. If the result of this and is zero, the line of code is skipped. The way this works is this:

Let $r1 = 1 and $r2 = 2

```
cmp     $r1, $r2
⇒ 100 because 1 < 2
add     $r3, $r4, 110
⇒ the add completes because 1 <= 2
```

Different registers are used for the add command because the registers used for the comparison aren't necessarily what's going to be changed in the next line.
The condition code is anded with the condition code from the instruction. If they are zero, the instruction does not complete because the statement (1 < 2, etc) isn't what the command was looking for. If the anded code is not zero, then there was a match somewhere (1 <= 2 is true because 1 < 2 or 1 == 2).
If a condition code is not specified in the command, it is set to the default of 111, which executes the command no matter the CPU's condition code.

Explanation of Load Memory Bit (LM)

LM bit enable our processor to load data from the memory and perform operation on it immediately. With the help of LM, memory read takes much less than a instruction. In MIPS, lw is one instruction and takes multiple cycles to perform. However, our processor can load from memory with an extra cycle.

As for array, access and write. Both MIPS and our processor needs to move the array pointer step by step, but our processor combine read memory and another instruction, saves a few clock cycles per memory read.

However, as for struct, MIPS has the advantage of access memory with a fixed offset. However, our processor would need to perform an addi to access specified data. This is kind of a drawback without offset immediate as in MIPS.

The actual reason of leaving out offset immediate area is that we only have 16-bit instruction. If we specifies two register in lw, there are only 4-bit left for the immediate. 16 Word offset is way too tiny. MIPS has 32-bit instruction, so it has plenty of spaces for immediate.

The Stack Pointer Register

The rtl design become pretty tough with $sp within the register file. For push and pop, we need to read or write the memory while updating $sp, but we need to use the interface of register and perform similar operation every time. We decided to create a totally separate register for $sp with easy interface for incrementing and decrementing, it it going to make push and pop rtl design so much easier and a few cycles fast.
The stack pointer register $sp is no longer accessible by the user because it does not need to be accessed. After the commands push and pop were added, the stack pointer does not need to be available to the user. It is no longer one of the sixteen main registers, and has been replaced with register s3.

Decision about exceptions and interrupt

Though we have plan eret for our processor, when it comes to design the rtl for eret, things get too complex to be straight out. Previous plan for restoring $t0 and put exception register as argument to exception handler will enable us to handle exception and return back to the program. However, after reconsidering the project requirement, we realized that exception handler is not really required. We do need ‘interrupt’ to provide input, call a certain function and display the result. For those interrupts, our CPU is not actually restarting from where exception happens, but jumps to a new locations and starts to perform new task. Those input I/O interrupts always abort the previous program and start a fresh new one. Therefore, for our project, we can do not really need to handle the exception, but simply reset CPU into a fixed initial state (given exception handler).
We decide to design and implement our CPU without exceptions. Later on in the term, we can simply add a few entrance to reset CPU to a fix state to perform the I/O interrupts required.

Milestone 3

How to enable our processor to instantiate arrays and objects

After the last meeting, we realized that our processor can not do anything involving a pointer like an array or an object, because there is no way to get an address to stack space. We think that being able to do array and objects are very important.
If we let the stack pointer be visible again, we will make push and pop slower. Right now they are even faster than regular calculation instructions like add and and. Fred came up with the idea of two stacks -- data stack and procedure stack.
The procedure stack keeps track of return address, restored ‘s’ and ‘t’ registers, while the data stack keeps track of arrays and other data that needs a reference. No pointer is allowed to point to any address within the procedure stack.
There is one ‘dp’ register. This register keeps track of the data stack. We can add a big value to it to create a big chunk of memory for an array. If a programmer wants to dereference some variable and use its address, we will increment dp by certain amount, and store the variable there. All pointers in the program have to point somewhere within data stack or heap.

The advantage of two-stack design is that it is immune from buffer overflow attack because return address and any array will be in two different stacks and no one can modifier return address in procedure stack. Also, it keeps our processor fast. We no longer have to update value of stack pointer every time we push or pop a value. Also, for function without allocating

local addressable memory, we can save two instructions for increment and decrement data pointer.
The disadvantage of two-stack design is that we are giving more pressure on operation system to organize memory since we know have an extra chunk of memory needed to be kept separate.

Changes to the RTL

We changed the RTL to be more like the ones shown to us in class. We adjusted how it was displayed from showing cycles in columns to showing them in rows, and we simplified all of the R-types to be one column. However, all other commands still needed their own columns in the RTL. We also simplified the RTL by removing all blank middle cells and moving up commands, even if it means we can't name all of the cycle anymore. To save room in our Design Document, we didn't name the cycles at all.

Changes to Branches and Jumps

Branches and jumps had a lot of redundancy, and only in special use cases were they different. Our group changed branches and jumps to remove this redundancy.
We decided that we only needed two commands: jump register (jr) and jump label (jl). Jr became an R-type instruction and is now storing the register that it receives in the second register slot instead of the first, like so:

jr	empty register	$r1	empty	condition code

Branches were removed entirely, to be replaced by jl. Jl jumps to a label if the condition code is satisfied, but the label is a sixteen bit address. This is done by using thirty-two bits of instructions for each jl/jal. Each time a jl is read into the instruction register, the control Unit signals that the next line needs to be read in as the address to jump to. Then the condition code and whether to set $ra is evaluated. B-type instructions were removed completely.
Jl vs jal is set using one bit, rather than the five that were used for jop before, and a reference to $ra is passed if the command is jal.

Moore Machine Vs Mealy Machine

We decide to implement a mealy state instead of moore machine, so the control signal output depend not only on the state, but also the input signals. With this method, one control state can provide different control signals and different instructions can perform different work at Cycle 2. Though we might need to wait control to produce those bits, these time will not be very long, but

we can save a whole cycle. If we start branch at cycle 2, we can make jl, jal, lui, cpi, push and pop one cycle faster.

Creating Tests

To create tests, we used a table format with one column listing a name and another listing a description of the test. For the unit tests, we decided to specify only units which we are going to implement rather than trying to test the xilinx/resources provided Muxes and other components. Our process for these tests was to list all inputs and assumptions then check the output using specific, immediate values. For the testing of the datapath, we decided to implement tests for unique cycles for instructions. For example, we only had one test for cycle 1, since there is only one unique type for all instructions. For cycle 2 there are multiple types of cycles.

Milestone 4

Control Unit design

Control Unit is a finite state with multiple different outputs. We have decided that we are going to use a mealy machine instead of a moore machine, so each state can have different control signals output and each instruction can start performing different task in cycle 2.
With mealy machine design, we do not have to create a state for every instruction. If certain instructions are doing very similar things in a cycle, they can have the same state to simplify the state design. For example, cycle 3 of jal, j and jr all update PC to a branch address, but jal also needs to write PC into register file. Instead of making three different states, we can just combine them into one, and set RW depending on whether it is jal. Therefore, we are able to shrink the number of states to 9.
There are many options we have to implement such a FSM. We can either use a 4 flip flop and represent them with binary encoding. This way we can minimize the number of flip flop we have but also make transition logic more complex and less intuitive. In our project, the number of flip flop is not as important as ease of design and speed. We decided to use one hot design. We have 9 flip flop, one for each state. At any time, only one of those 9 flip flop can contain 1. This way we can convert FSM transition into machine level logic directly and intuitively. Though we would need more flip flop, state transition is faster and easier to design. For those output, we can take advantage of lots of “don’t care”. For example, if RW is 0, we don’t care what RWSrc is because its output is going to be thrown away. With the help of that, we need less logic gates and be able to reduce transition time. Each control signal output has a table that specifies state, input value and corresponding output.

control Unit Xilinx implementation

Datapath is easy to implement with schematic because it is very intuitive to see blocks. However, it wouldn't be easier to implement control Unit with schematic, because there are going to be overwhelming gates and the diagram would be super messy.
In contrast, Verilog is a very powerful and useful tool to design hardware. With behaviour level Verilog, we can describe control Unit's behavior and ask Verilog to generate corresponding hardware for us. We can use "if", "else", "case" to "program" hardware. It would much more easier to implement. Fred actually wrote and tested a behavior-level Verilog of control Unit overnight. However, behavior level Verilog cannot take advantage of "don't care" bit. It can generate hardware that meets the requirement, but it also meets many unrequired additional conditions. Verilog is going to produce more hardware to fix values that we don't actually care.
In our Xilinx file, behavior level control Unit is "ControlUnit.v", and there is another file "ControlUnitFast.v". Instead of descripting circuit behavior with "always block", we will implement another control Unit at register-transfer level. In other word, we will draw out K-Maps for each control signals, and find the optimal logic design manually and then implement them with continuous "assign" in Verilog. This way we can take full use of "don't care bits" and make our control faster!

Testing of control Unit

The testing of control Unit can be a bit tricky, since there are so many cases of control Unit. It would be super hard to cover everything. Also, it is going to take a long time to write every cases. One alternative way is create a ControlUnitTester module, which will take state, input to controls and output corresponding control signals and next state. For control signals that can be don't care, ControlUnitTester would output an extra bit to specify whether this signal is "don't care". ControlUnitTester is implemented simply by copying every states in FSM. As long as programmer do not make typo, it would be correct. ControlUnitTester is all logic gates without storage, and serves as a standard answer producer machine. **In test bench, we can run ControlUnit with specified input combination for multiply cycles and check its output and transition to ControlUnitTester.** The checking also cover the case of "don't care". In actual code, for example, "MSrc" is the output from ControlUnit, and is compared with the standard answer "MSrca" from ControlUnitTester. If they equal or "MSrca" indicates "don't care", then "MSrce" is going to be 1 to indicate that "MSrc" pass the test in this case. If any control signal do not pass the test, "error" would be high. Sometimes, our FSM would transit to some impossible state, with all control signals to be "don't care". We do not want to miss those errors, so ControlUnitTester also output an "error" bit to signal that the combination of input and states is not supposed to happen.

ALU Design

Our current ALU is designed with Verilog with a very simple behavior level description. Overflow is checked after we have the answer. We are not sure whether Verilog would automatically generate carry-look-ahead for us. Our ALU might be using the slow carry-ripple adder. Also, the overflow checking is very slow, since it needs to wait until adder finishes its job.
This design is not perfect, but we decided to move on with it. If we have more time latter, we will work on redesigning it in a lower level with carry-look-ahead.

Exception and I/O handling

For this milestone, our datapath is not able to do I/O handling and exception yet. We will fully test the core part first and add exception handling later. All we would need to is to add a RESET control signal to set PC to a specified address and move stack pointer to its original position. Whenever an exception happens, for example overflow or memory access error, our CPU would abort the current process and restart from a known process. If there is an I/O interrupt, we will set PC to a specific handler to handle this kind of I/O, for example read data, call relPrime().

In order to fulfill the requirement of I/O, there will be another control and even FSM at top level outside control unit and datapath. These controls are not general purpose, solely designed for this project. We will start work on Exception and I/O handling in next milestone.

Fixing Integration Tests

After the last milestone, we realized the integration tests we designed did not fulfill their intended purpose. For this milestone we fixed this problem by rewriting the tests. We tried to test every connection between two components by sending through inputs and reading outputs. For instance, we made sure the pc register and adder would only increment the pc's value when pc write was set to 1. We did not consider edge cases in the design of the tests because they were covered during our unit tests.

Milestone 5

System testing

We add four muxes into our datapath to form datapath Test. Those muxes enable us to directly load and check data in register and memory. For system testing, we don't need to rebuilt memory with coe file for each test cases. We can just load instructions code in test bench --

```
run(startPC, numInstru)
readMem(address, out)
readReg( address, out)
writeMem( address, out)
writeReg( address, out);
```

Those testing interface make system test much easier.

Get rid of state 9

Everything is almost complete. We are looking for what we can improve. As we go through what our datapath is doing for each cycle, we realized that write back cycle is basically doing nothing compared to others. ALUO is only used to write data back to register in our design, and it only takes a very short amount of time for ALUO data to reach RWD of register file. The write back cycle is basically waste of time and resources.

Therefore, we decided to get rid of state 9 and combine write back and execution cycle of R-type and addi. In other word, the output from ALU won't be stored for a cycle, but directly written into register file. This makes our R-type and addi only 3 cycles long without compromising any clock frequency.

This change does not take a lot of time, since all we need to do is to delete everything related to ALUO.

Synchronous Design:

We decide to modify memory and register file to make them synchronous. In other words, memory and register file only updates its output at clock edge. Once the data are updated, they aren't going to change within the same cycle period. Synchronous component can been seen as asynchronous component with its output connected to flip-flop, which only update at rising clock edge. Since register file is already synchronous, we no longer need to store its output in some register temporarily. So we get rid of register A and B. Besides that, we also make some big modification in order to make memory synchronous and operate in only one cycle. Those changes including making instruction register asynchronous and un-clocked and add an delayed mux are discussed in the next section.

Memory modification

Memory is pretty complex and slow. We will use a block memory to save more resources for other component. However, block memory is slower. We designed our RTL assuming that we have an asynchronous memory unit in the first time. By the time we realized that asynchronous memory is going to take too much resources, and we won't be able to implement our CPU in the board, it is already 9th week. We don't want to make too much changes up to this point.

The first option we have is of course, wait memory for an extra cycle. This is going to introduce an extra cycle, that is basically doing nothing -- write MD into Instruction Register. Also, we would need to add new states and make changes to our control. Making huge design changes in a late time is not a very smart choice. This option involves a lot changes in control and will slow our processor by around ⅓ .

The second option is to add an inverter in front of the clock of memory. Therefore, when other components experience a falling clock edge, memory experiences a rising edge and output the read data. Therefore, we don't need to wait for another cycle for the rising clock edge, and memory access can still be done within one cycle. However, this is actually makes execution time even longer, since memory read only at the first half of the cycle. If we give memory two cycle, we only add one extra cycle per instruction, but if we only use half cycle to read from memory, our clock cycle is actually going to double. This option won't introduce extra design work, but will double the execution time.

The third option is to delay the memory clock in the second option even more. We can delay the rising edge of memory to be ¾ cycle later than others, so it will use most of the cycle to perform read and we only need to slow our processor by a smaller amount, ⅓ in this case. We can even delay the clock edge ⅞ cycle or even more, and reduce the clock cycle. The way we can divide cycle time is to increase clock frequency and used use a counter to create slower cycle and phase shift. For example, if we originally have 100MHz clock and we want to introduce a ⅞ cycle delay in memory. We can first set primary clock to 800MHz, which controls a 3-bit counter. The counter would go from 0, 1, 2, up to 7 and then back to 0 each cycle. When the counter is between 0 and, it output 1, otherwise it outputs 0. This way we can generate a 100MHz clock from a 1600MHz clock. For the ⅞ delayed clock, it output 1 when counter is at 7, 0, 1, 2. In this way, we can get two clock we a ⅛ cycle difference. This option does not involve any changes in control, but we would need extra flip flops for counter. The other drawback for this option is power. If we double the clock frequency, it consumes 4 times more energy.

The last option or what we eventually implement is to give memory a whole cycle and make modifications to components involved with MD. The Xilinx built-in block memory is synchronous. We may see it as an asynchronous memory with a temporary register at its output. Instruction register and mux for B are modified.

Instruction register is changed to a 16-bit latch, no longer controlled by the clock. When its EN is high, instruction register would update its data regardless of clock. Therefore, read instruction can go in and out of instruction register in one cycle. However, the control unit will need to be modified to accommodate this change. In the old design spill out IW = 1 in state 1, but now it needs to have IW = 1 in state 2 to update its data. We need to delay IW. This is easy

to do, since all instruction transit from state 1 to state 2. We can easily make control unit output IW = 1 at state 2.

Besides instruction register, B register is also using data from MD. Similarly, we need to do some minor modification for B register. Since MD is read one cycle later, we need to delay LM for only cycle to select whether we need B to be Reg[r2D] or MD. This is also easily achieved with a delayed mux. Delay mux would delay the control signal by one cycle with a flip-flop. So we don't need to modify complex logic of LM in control unit.

We update the datapath diagram in the design document, and Xilinx implementation. The new design works well and pass all RTL tests.