Super Awesome Processor A (SAPA) v1.0

Load-Store/Accumulator

Team 1A - Khaled Alfayez, Shaun Davis, Trinity Merrell, Logan Smith

CONTENTS:

Executive Summary

SAPA v1.0 processor is a new, state-of-the-art processor developed in Terre Haute, IN. This processor is primarily intended for non-robust coding projects, and offers options for easy-to-understand instructions, competitive program sizes, and lightning-efficient processing speeds. SAPA v1.0 has provided correct results for relPrime algorithms for one month and standing.

This report reviews SAPA's 12 official assembly instructions as well as their machine language translations. Beyond instructions, this report outlines everything necessary to understand the SAPA processor from the inside out. This includes everything from implementing and testing individual units to integrating the processor entirely.

Our target audience includes programmers of all ages, though particularly those with a penchant for the software and hardware realms of computer architecture.They know the value in a powerful processor and have a solid idea of the kind of efficiency necessary to carry out computations of all sorts.

Our processor edges out competition in two areas: ease-of-use and performance. All of our instructions are sized at 16-bits, allowing for easy to understand machine language translations. We also offer users an assembler capable of translating a host of pseudoinstructions available for convenience. Performance-wise, our processor runs faster than the average processor in 1990 at about 60Mhz. This is easily on par with many competitors on the market today.

The long-term goal of the SAPA v1.0 is to provide programmers a simpler processor design that is not only easy to implement and adapt, but efficient and accurate. The hope is to change the industry to see processors as not only tools but machines that are beautiful and adapting with every day.

Introduction

Our processor, appropriately named the Super Awesome Processor A (SAPA) v1.0, is a 16-bit processor based off of MIPS 32-bit architecture. Most of the differences are attributed to the Load-Store/Accumulator design, accomadations for a multi-cycle datapath, and the reduced 16-bit size. Our design has three different types of instructions that allow us to compute arithmetic, jump to other instructions, and use immediate values. Additionally, multiple pseudo-instructoins are available to the user for convienience. There are sixteen registers implemented by our design, outlined in the Appendix.

To achieve a completed processor, out team had to make many design decisions, involving compromise and some sacrifices. We originally had twenty registers for use, but trimmed that down to better fit the 16-bit size of our processor, giving us less flexibility when writing code. Our choice in architecture – Load/Store with Accumulator, caused us to have many extra muxes to choose the value from our special computation result register as well as a special register in the datapath that always stores the value of the computation result register. All of this was caused by our accumulator computation register.

Because our design is also load/store, we found it necessary to have a memory unit as well as a register file to hold data values. Between these two components and our ALU, we found it necessary to have special temporary register such as IR, MDR, A/B/C

to hold necessary values between stages in the datapath. Because our team decided to implement a computation register, while it was convenient, we had to be careful when considering how to write our instructions.

Overall, we believe our processor has an intuitive desgn that takes advantage of both the load-store and accumulator arhcitectures.

Instruction Set Design

Our processor utilized three different types of instructions, all 16-bit. First, we have the C-type instruction which handle register to register computations for arithmetic and logic. Within this type, there are instructions such as *add, sub, and, or, etc.* These instructions take two registers as operands to their computations, share a single opcode, and are distinguishable by their func code. Their results are always stored in the specialized computation register.

The second instruction type, I-type instructions, are used for register to data and register to memory computations. These instructions handle storing data from registers into memory and loading data into registers from memory, immediate values, and the computation register. They also handle control flow such as branches and jumps that are necessary for loops and procedure calls.

Lastly, we have the L-type, or leap type, instructions which are used for jal and j instructions. Jump instructions use a 12-bit immediate with the top 4 bits of the PC concatenated to create a 16-bit address and allowing for a 2^11 size block of “jump space”. We made this instruction type instead of just using an I-type again because we wanted more range for jumps.

Implementation

According to our implementation of the SAPA v1.0 processaor, only one instruction can go through our instruction pipeline at a time. We have also utilized multi-cycle register transfer language, which means each instruction is executed in 4 or more steps.

In order to implement this multi-cycle processor, we needed to develop a control unit that handled both fetching new instructions from memory and moving through the appropriate cycles when a new instruction is sent into the processor.

For each cycle, different control bits were sent to each unit in the datapath such as write and read controls on registers and the memory unit, and the op code for the ALUControl unit **(see A1. Specficiation for Controls)**.

Due to our load-store/accumulator architecture type, the code density of our processor is relatively high. Since data is held in registers, the tradeoff for high density programs are programs of smaller sizes. Since each instruction is equal in size, programs are much easier to store.

Xilinx Model

For our different units, we made them using Verilog and schematic. It was pretty easy moving between both meathods for debugging purposes and made making the units easy for everyone.

For implementing the datapath however, we kept it in schematic since it is much easier to take all of our previous work, moving it over and connecting it all together rather than trying to retype in all of the things we have already done.

This processor can be made in either all schematic or all Verilog but we avoided doing this since it would limit our productiveness if the person working on the specific file wasn't proficient in what was being used.

Testing Methodology

Our testing strategy consisted mostly of trying to anticipate problems before they occurred. We think of what might go wrong with a component and write tests to catch those situations. After that, we write general cases for expected outputs with non-extreme or expected data. When these tests are finished, we run through all inputs that satisfy the tests and make sure the output is expected/acceptable.

Testing individual units was fairly simple since all they needed to do was fulfill whatever function we designed them for and we are all pretty cabable at using Verilog so if anything went wrong, it was just a little debugging.

When we started combining units together to make parts of the processor, we decided that it would be a good idea to have people make parts of the datapath that they had no influence on the units that are used if possible. This is so that the unit actually works and it's not just the person who designed it inserting magic numbers or using functions that the unit isn't supposed to have. This is also a way for us to insure that we all understand the datapath we made since it should as easy as a very advanced connect the dots in which you just insert the appropriate units and connect them all together.

One problem we had while testing the parts was that sometimes what we wanted to test wouldn't show the result until the cycle after execution. It was quickly realized and

then solved by having the parts update on the negative clock edge rather than the positive edge.

We had this same problem while testing some of our functions for the datapath but we couldn't use the same solution since we wanted it to all trigger on the same clock edge to reduce confusion. We figured out how to input stalls without messing with the datapath in order to check our functions.

Final Performance Results

Space

The relPrime algorithm detailed in **A1. relPrime (Machine Language)** is a total of **200B** for Euclid's algorithm and **614B** for the relPrime algorithm in total. These figures were determined by the equation *n * 2 + m * 2 = totalBytes,* where n is the total number of instructions and m is the total number of independent memory instructions (ie instructions where memory accessed to be either store or loaded). We recognize this is is large compensation of space but, as mentioned previously, this is due to the architecture of the processor.

Time

In testing, the SAPA v1.0 clocked a cycle time of **58.431 Mhz** - faster than the average processor from the early 1990s, which was 50Mhz. After running the algorithm for relPrime detailed in **A1. relPrime (Assembly)**, with the previous condition condition n = 0x13B0 (where n = $a0 register), **36,360 instructions** are run over **40,099 cycles**.This produces **1.09 average cycles per instruction**. On the processor, the execution time is **7.35558ms**.Overall, these results are not the most inefficient and we feel our processor is on average, at the level of competitors in this area.

Conclusion

Overall, we believe that our processor has a solid and user friendly design. To reach a finishing point point for this project, our team faced many challenges and late nights. However, what we learned from this project was invaluable. We are prouid of how we implemented this processor to leave room for adaptability due to the wide range of instructions. Being more efficient from computers from the 90's, while it is not up to today's standards, it is very functional and a good working processor.

Appendices

CONTENTS:

Register Descriptions

Table of Register Descriptions

Register	Number	Availability	Description
$s*n* *(0-2)* <(^.^)>	0-2	Read/write	General purpose registers. The intent is to store long term computation results and save values over functions calls
$t*n* *(0-3)* <(-_-,)>	3-6	Read/write	General purpose registers, to be used like $s*n* registers. Will NOT be saved over function calls.
$v <(-_-,)>	7	Read/write	This register is for storing the return value from a function
$cr <(-_-,)>	8	Read only*	Accumulator - stores most recently computed value
$ra <(^.^)>	9	Read/write	Stores the return address of a function
$a*n* (0-1) <(-_-,)>	10-11	Read/write	These registers are used to store arguments for use in a called function
$st <(^.^)>	12	Read/write	Reference “top” or lowest memory address of stack
$i*n* (0-1) :~/	13-14	Not available	Used by assembler for pseudo instructions.
$k :~/	15	Read/write (while handling exceptions)	Exception registers; only accessible with permissions*

Unsafe between procedure calls = <(-_-,)> **Safe between procedure calls = <(^_^)>**

*** Enforced by exception handler**

Instructions - Assembly and Machine Language

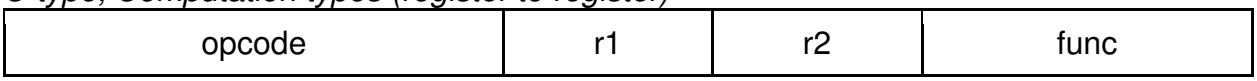
Procedure Call Conventions:

1. Registers $i*n*, $k*n* (where n is 0-1) are reserved for the assembler and operating system and should not be used by user programs or compilers.
2. Registers $a*n* (where n is 0-1) are used to pass arguments to procedures, any other arguments should go on the stack. Register $v is used to return a value from functions.
3. Registers $a*n*, $t*n*, $d, $cr, and $v are temporary and volatile. Expect them to contain different data after a procedure call.
4. $s*n* registers must be backed up on the stack at the beginning of a procedure and restored before returning from the procedure. This preserves values in these registers over procedure calls.
5. $st is the stack register. It points to the top memory location in the stack. If the stack is grown at any time in a procedure, it must be reduced before returning from that procedure.
    a. Memory is allocated to the stack by subtracting from the value in $st. Memory is deallocated from the stack by adding to the value in $st.
6. $ra is the return address of a procedure. Jal will overwrite $ra to be the next instruction, so $ra must ALWAYS be backed up on the stack before a procedure call and restored after returning from the procedure.
7. The instruction jr $ra will return the program to the address.

Syntax and Semantics:

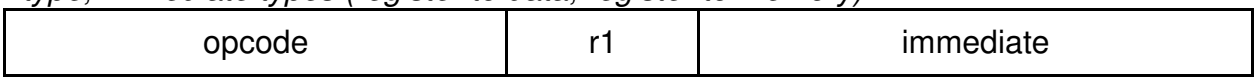
Basic Instruction Formats

C-type, Computation types (register to register)



C-type instructions are used for register to register computations. They handle arithmetic and logical computations such as add, sub, and, or, etc. These instructions share a single opcode, and are distinguishable by their func code.

I-type, Immediate types (register to data, register to memory)



I-type instructions are used for register to data and register to memory computations. These instructions handle storing data from registers into memory and loading data into registers from memory, immediate values, and the $cr register. They also handle control flow such as branches and jumps that are necessary for loops and procedure calls.

L-type, Leap type

opcode	immediate

15 11 0

L-type instructions are used for jal and j instructions.

Arithmetic and Logical Instructions

Arithmetic and logical instructions are **C-type** instructions. These instructions take two registers as operands to their computations. Their results are always stored in the specialized $cr register.

Arithmetic and logical instructions are directly translated from their assembly to the machine language. For example, the following assembly would translate accordingly into binary:

add $s2, $t3

0000	0010	0111	0000
op	r1	r2	func

Registers are directly translated from their respective numbers in the registry (see chart in Registers).

List of Arithmetic and Logical Instructions:

Addition:

add r1, r2

0000	r1	r2	0000
4	4	4	4

Stores the sum of r1 and r2 into register $cr

Subtraction:

sub r1, r2

0000	r1	r2	0001
4	4	4	4

Stores the difference between r1 and r2 into register $cr

AND:

and r1, r2

0000	r1	r2	0010
4	4	4	4

Stores the logical AND of r1 and r2 into register $cr

A1. Design Documentation

OR:

or r1, r2

0000	r1	r2	0011
4	4	4	4

Stores the logical OR of r1 and r2 into register $cr

NOR:

nor r1 ,r2

0000	r1	r2	0100
4	4	4	4

Stores the logical NOR of r1 and r2 into register $cr

NAND:

nand r1, r2

0000	r1	r2	0101
4	4	4	4

Stores the logical NAND of r1 and r2 into register $cr

Exclusive OR:

xor r1, r2

0000	r1	r2	0110
4	4	4	4

Stores the logical Exclusive OR of r1 and r2 into register $cr

Set Less Than:

slt r1, r2

0000	r1	r2	0111
4	4	4	4

Stores either a 1(True) or a 0(False) in $cr depending on if r1 is less than r2

Branch Instructions

Branch instructions are **I-type** instructions, and compare the value in $cr to the value in r1. The branch instruction will then succeed or fail based on equivalence or inequality.

Branch instructions are PC-relative, meaning they use an 8-bit offset that allows a user to jump 2^7-1 instructions forward or 2^7 backward. A translation may appear as below (where “loop” is 3 instructions above bieq):

bieq $s0, loop

0001	0000	1101
op	r1	BranchAddr

A1. Design Documentation

List of Branch Instructions:
Branch if equal:

bieq r1, location

0001	r1	BranchAddr
4	4	8

Conditional branch to address in immediate if r1 is equal to register $cr

Branch not equal:

bneq r1, location

0010	r1	BranchAddr
4	4	8

Conditional branch to address in immediate if r1 does not equal register $cr

Jump Instructions

Jump instructions utilize two instructions formats: **L-type** and **I-type.**

Jump instructions use a 12-bit immediate with the top 4 bits of the PC concatenated to create a 16-bit address and allowing for a 2^{11} size block of "jump space". PC is set to this new address.

List of Jump Instructions:
Jump:

j location

0011	JumpAddr
4	12

Unconditional jump to the address in immediate

Jump and link:

jal location

0100	JumpAddr
4	12

Unconditional jump to the address in immediate, storing the address of subsequent instruction into register $ra

Jump register:

jr r1

0101	r1	unused
4	4	8

Unconditional jump to the address in r1

Load/Store Instructions

Load/Store instructions are **I-type** and often require a value to be computed and stored in $cr prior to execution. Specific requirements are denoted for each instruction.

Load/store instructions are directly translated similar to arithmetic and logical instructions. The following assembly would translate accordingly:

lw $t0

1010	0101	unused
op	r1	Imm

List of Load/Store Instructions:

Load upper immediate:

lui r1, upper(big)

0110	r1	Immediate
4	4	8

Load top half of 16-bit immediate into r1. Bottom bits are set to zero.

Load lower immediate:

lli r1, lower(big)

0111	r1	Immediate
4	4	8

Load lower half of 16-bit immediate into r1. Top bits are set to sign of immediate.

Load to register:

ltr r1, immediate

1000	r1	SignExtImm
4	4	8

Take a sign-extended 8-bit immediate value and store in r1. If immediate is greater than 8-bits, utilizes load upper immediate and load lower immediate.

ltr r1, big

Translates to:

```
lui     $i0, upper(big)
lli     $i1, lower(big)
or      $i0, $i1
ctr     r1
```

Copy to register:

ctr r1

1001	r1	unused
4	4	8

Take a previously computed value from $cr and store in r1

Load word:

lw r1

1010	r1	unused
4	4	8

Access memory at the address stored in $cr and store the value in r1

Store word:

sw r1

1011	r1	unused
4	4	8

Store value in r1 at the previously computed memory address in register $cr.

Special Procedures

Interact with I/O

syscall

1100	unused
4	12

How to use syscall:
Step 1. Load appropriate code into $v.
Step 2. Load argument values, if any, into $a0 or $a1 as specified.
Step 3. Issue the SYSCALL instruction.
Step 4. Retrieve return values, if any, from result registers as specified.

Example:

```
ltr     $v0, 1      # service 1 is print integer
ltr     $a0, 0      # load desired value into argument register $a0
syscall
```

A1. Design Documentation

Table of Syscall Codes

Function	Integer in $v	Argument or Return Value
PRINT_INT	1	$a0 = value
PRINT_STRING	2	$a0 = address of string
READ_INT	3	Result placed in $v
READ_STRING	4	$a0 = address, $a1 = maximum length
EXIT	5	None
PRINT_CHAR	6	$a0 low byte = character
READ_CHAR	7	Character returned in low byte of $v

Pseudo Instructions

These are instructions are not official instructions but sets of instructions that our processor will handle with smaller instructions. All pseudo instructions are **I-type**. It is important to note that the expansion of pseudo instructions often require **C-type** instructions. Therefore, users should generally assume that $cr will be overwritten when executing pseudo instructions and backup $cr if they wish to use it afterward.

Set branch comparison:

sbc imm

Sets register $cr to a sign-extended 8-bit immediate value in preparation for a branch instruction.
Translation:

```
ltr   $i0, 0
ltr   $i1, imm
add   $i0, $i1
```

Set jump register:

sjr JumpAddr

Sets register $ra to a jump address specified by the user.
Translation:

```
ltr   $i1, JumpAddr
ltr   $i0, 0
add   $i0, $i1
ctr   $ra
```

A1. Design Documentation

OR immediate:

```
ori     r1, imm
```

Stores the logical OR of r1 and immediate into register $cr. 8-bit numbers are zero-extended.
8-bit Translation:

```
lli     $i0, imm        # Loads zero-extended lower 8 bits
or      r1, $i0
```

16-bit Translation:

```
lui     $i0, upper(imm)     # Loads upper 8 bits, bottom bits filled with 0
lli     $i1, lower(imm)     # Loads zero-extended lower 8 bits
or      $i0, $i0
or      r1, $cr
```

ADD immediate:

```
addi    r1, imm
```

Stores the arithmetic result of r1 and immediate into register $cr. 8-bit numbers are zero-extended. For 16-bit numbers, the following translation occurs:
8-bit Translation:

```
ltr     $i0, imm
add     r1, $i0
```

16-bit Translation:

```
lui     $i0, upper(big)
lli     $i1, lower(big)
or      $i0, $i1
add     r1, $cr
```

Load address:

```
la      r1, address
```

Stores the sign-extended address in immediate into r1.
Translation:

```
lui     $i0, upper(address)
lli     $i1, lower(address)
or      $i0, $i1
ctr     r1
```

Register to Register:

```
rtr     r1, r2
```

Moves the value in r2 to r1.
Translation:

```
ltr     $i0, 0
add     r2, $i0
ctr     r1
```

A1. Design Documentation

Table of Machine Language Translations for Instructions

Key: R[8] = $cr, R[9] = $ra

Instruction	Type	Code	Description of bits and rules			
Add	C	R[8] = R[r1] + R[r2]	0000	r1	r2	0000
Sub	C	R[8] = R[r1] - R[r2]	0000	r1	r2	0001
AND	C	R[8] = R[r1] & R[r2]	0000	r1	r2	0010
OR	C	R[8] = R[r1] | R[r2]	0000	r1	r2	0011
NOR	C	R[8] = ~(R[r1] | R[r2])	0000	r1	r2	0100
NAND	C	R[8] = ~(R[r1] & R[r2])	0000	r1	r2	0101
XOR	C	R[8] = R[r1] ^ R[r2]	0000	r1	r2	0110
Set Less Than	C	R[8] = (R[r1] < R[r2]) ? 1 : 0	0000	r1	r2	0111
Branch Equal	I	If (R[8] = R[r1]) PC = PC + BranchAddr	0001	r1	BranchAddr	
Branch Not Equal	I	If (R[8] != R[r1]) PC = PC + BranchAddr	0010	r1	BranchAddr	
Jump	L	PC = JumpAddr	0011	JumpAddr		
Jump and Link	L	R[9] = PC + 1 PC = JumpAddr	0100	JumpAddr		
Jump Register	I	PC = R[r1]	0101	r1	8'b0	
Load Upper Immediate	I	R[r1] = {imm, 8'b0}	0110	r1	Immediate	
Load Lower Immediate	I	R[r1] = {8'b0, imm}	0111	r1	Immediate	
Load to Register	I	R[r1] = SignExtImm	1000	r1	Immediate	
Copy to Register	I	R[r1] = R[8]	1001	r1	8'b0	
Load Word	I	R[r1] = M[R[8]]	1010	r1	8'b0	
Store Word	I	M[R[8]] = R[r1]	1011	r1	8'b0	

syscall	I	I/O	1100	4'b0	8'b0
Set Branch Comparison	-	R[8] = SE(BranchComparison)	*Pseudo instruction*		
Set Jump Register	-	R[9] = JumpAddr	*Pseudo instruction*		
ORi	-	R[8] = r1 | ZeroExtImm	*Pseudo instruction*		
Load Address	-	R[r1] = SignExtImm	*Pseudo instruction*		
Register to Register	-	R[r1] = R[r2]	*Pseudo instruction*		
Addi	-	R[r1] = SignExtImm	*Pseudo instruction*		

Table of Common Operations:

	SAPA 1.0	Description
Load Address	*la $s2, 0x4EF6*	Loading an address is a pseudo instruction. Refer to *Pseudo instruction* for full details on how *la* works.
Arithmetic / Logical	*ltr $t1, 1* *ltr $t2, 1* *add $t1, $t2*	Load values into two registers and call add to store their sum in $cr. Similar for other arithmetic and logical operations.
Iterations	*…* *ltr $s1, 15* *ltr $t0, 1* *ltr $t1, 0* *ltr $t2, 3* *loop: slt $s1, $t1* *bieq $t0, exit*	This *for-loop* keeps adding 3 to register $s3 (0) until $t1 becomes greater than $s1 (15), resulting with a 18 stored in $t1 after the loop has exited.

	add $t1, $t2 *ctr $t1* *j loop* *exit: ...*	
Branches	*See above*	The example above utilizes *slt* and *bieq* to create a branch on greater than, which isn't an instruction itself but can be created using multiple instructions.
Jumps	*…* *add2: ltr $t0, 2* *add $t0, $a0* *ctr $t1* *sbc 20* *bneq $t1, add2* *sjr end* *jr $ra* *…* *end: …*	This example of code does recursive addition of adding two until the value of 20 is reached. However, it needs to move to end instead of moving to the next set of recursive code. Unfortunately, it is too far for the regular jump code to reach. The user can instead; however, set the value of the desired address to the $ra register and can now jump to it.

relPrime and Euclid's Algorithm (Assembly):

```
relPrimeSetup:
        addi    $st, 0          # $cr = $st
        sw      $ra             # store $ra on stack
        addi    $st, -4         # $cr = $st - 4
        ctr     $st             # $st = $st - 4
        sw      $s0             # store $s0 on stack
        addi    $st, -4         # $cr = $st - 4
        ctr     $st             # $st = $st - 4
        sw      $s1             # store $s1 on stack
        rtr     $s0, $a0        # $s0 = n
        ltr     $t0, 2          # m = 2
        rtr     $a1, $t0        # $a1 = m
```

```
        rtr     $s1, $t0        # $s1 = m
relPrimeLoop:
        jal     gcd
        sbc     1               # $cr = 1
        bieq    $v, cleanup     # if gcd(n, m) == 1, branch to cleanup
        addi    $s1, 1          # $cr = m + 1
        ctr     $s1             # m = m + 1
        rtr     $a1, $s1        # $a1 = m
        rtr     $a0, $s0
        j       relPrimeLoop
gcd:
        sbc     0               # $cr = 0
        bneq    $a0, gcdLoop            # if a != 0, branch to gcdLoop
        rtr     $v, $a1         # $v = b
        jr      $ra             # return to caller
gcdLoop:
        sbc     0               # $cr = 0
        bneq    $a1, subOne # if b != 0, branch to subOne
        rtr     $v, $a0         # $v = a
        jr      $ra             # return to caller
subOne:
        slt     $a1, $a0        # $cr = b < a
        ltr     $t0, 0
        bieq    $t0, subTwo  # if !(a > b) go to subTwo
        sub     $a0, $a1        # $cr = a - b
        ctr     $a0             # a = a - b
        j       gcdLoop         # return to gcdLoop
subTwo:
        sub     $a1, $a0        # $cr = b - a
        ctr     $a1             # b = b - a
        j       gcdLoop         # return to gcdLoop
cleanup:
        rtr     $v, $s1         # $v = m
        addi    $st, 0          # $cr = $st
        lw      $s1             # restore $s1
        addi    $st, 4          # $cr = $st + 4
        ctr     $st             # $st = $st + 4
        lw      $s0             # restore $s0
        addi    $st, 4          # $cr = $st + 4
        ctr     $st             # $st = $st + 4
        lw      $ra             # restore $ra
        jr      $ra
```

A1. Design Documentation

relPrime and Euclid's Algorithm (Machine Language):

relPrimeSetup:

addi $st, 0 # $cr = $st

1000	1101	00000000

0000	1100	1101	0000

sw $ra # store $ra on stack

1011	1001	00000000

addi $st, -4 # $cr = $st - 4

1000	1101	11111100

0000	1100	1101	0000

ctr $st # $st = $st - 4

1001	1100	00000000

sw $s0 # store $s0 on stack

1011	0000	00000000

addi $st, -4 # $cr = $st - 4

1000	1101	11111100

0000	1100	1101	0000

ctr $st # $st = $st - 4

1001	1100	00000000

sw $s1 # store $s1 on stack

1011	0001	00000000

rtr $s0, $a0 # $s0 = n

1000	1101	00000000

0000	1010	1101	0000

1001	0000	00000000

ltr $t0, 2 # m = 2

1000	0011	00000010

rtr $a1, $t0 # $a1 = m

1000	1101	00000000

0000	0011	1101	0000

1001	1011	00000000

rtr $s0, $t0 # $s1 = m

1000	1101	00000000

0000	0011	1101	0000

1001	0001	00000000

relPrimeLoop:

jal gcd

0100	000000101001

sbc 1 # $cr = 1

1000	1101	00000000

1000	1110	00000001

0000	1101	1110	0000

bieq $v, cleanup # if gcd(n, m) == 1 branch to cleanup

0001	0111	00100011

addi $s1, 1 # $cr = m + 1

1000	1101	00000001

0000	0001	1101	0000

ctr $s1 # m = m + 1

1001	0001	00000000

rtr $a1, $s1 # $a1 = m

1000	1101	00000000

0000	0001	1101	0000

1001	1011	00000000

rtr $a0, $s0 # $a0 = n

1000	1101	00000000

0000	0000	1101	0000

1001	1010	00000000

j relPrimeLoop

0011	000000011010

gcd:

sbc 0 # $cr = 0

1000	1101	00000000

1000	1110	00000000

0000	1101	1110	0000

bneq $a0, gcdLoop # if a != 0, go to subOne

0010	1010	00000100

rtr $v, $a1 # $v = b

1000	1101	00000000

0000	1011	1101	0000

1001	0111	00000000

jr $ra # return to caller

0101	1001	00000000

gcdLoop:

sbc 0 # $cr = 0

1000	1101	00000000

1000	1110	00000000

0000	1101	1110	0000

bneq $a1, subOne # if b != 0, branch to subOne

0010	1011	00000100

rtr $v, $a0 # $v = a

1000	1101	00000000

0000	1010	1101	0000

1001	0111	00000000

jr $ra # return to caller

0101	1001	00000000

A1. Design Documentation

subOne:

slt $a1, $a0 # $cr = b < a

0000	1011	1010	0111

ltr $t0, 0

1000	0011	00000000

bieq $t0, subTwo # if !(a > b) go to subTwo

0001	0011	00000011

sub $a0, $a1 # $cr = a - b

0000	1010	1011	0001

ctr $a0 # a = a - b

1001	1010	00000000

j gcdLoop # return to gcdLoop

0011	000000110001

subTwo:

sub $a1, $a0 # $cr = b - a

0000	1011	1010	0001

ctr $a1 # b = b - a

1001	1011	00000000

j gcdLoop # return to gcdLoop

0011	000000110001

cleanup:

rtr $v, $s1 # $v = m

1000	1101	00000000

0000	00001	1101	0000

1001	0111	00000000

A1. Design Documentation

addi $st, 0 # $cr = $st

1000	1101	00000000

0000	1100	1101	0000

lw $s1 # restore $s1

1010	0001	00000000

addi $st, 4 # $cr = $st + 4

1000	1101	00000100

0000	1100	1101	0000

ctr $st # $st = $st + 4

1001	1100	00000000

lw $s0

1010	0000	00000000

addi $st, 4 # $cr = $st + 4

1000	1101	00000100

0000	1100	1101	0000

ctr $st # $st = $st + 4

1001	1100	00000000

lw $ra # restore $ra

1010	1001	00000000

jr $ra

0101	1001	00000000

Register Transfer Language (RTL)

Definitions:

CR = Register[8]
RA = Register[9]
I0 = Register[13]
I1 = Register[14]
R1 = Register[IR[11-8]]
R2 = Register[IR[7-4]]

OP = IR[15-12]
IMM8 = IR[7-0]
IMM12 = IR[11-0]
IMM16 = 16-bit immediate
IR = Instruction Register
MDR = Memory Data Register

Common RTL:

Cycle	Label	RTL
0	FETCH INSTR:	PC = PC + 1 IR = Mem[PC]
1	DECODE:	A = R1 B = R2 C = CR ALUout = PC + SE(IMM8)
Last	DONE:	Goto FETCH INSTR

Unique RTL:

	Type C	Op 0000	func 0000 - 0111
Cycle	**Label**	**RTL**	
2	OPERATION:	ALUout = A op B	
3		CR = ALUout	

Instruction Branch if equal / not equal	**Type** **I**	**Op** **0001 / 0010**	**func** **-**
Cycle	**Label**	**RTL**	
2	BIEQ:	if (A == C) then PC = ALUout	
	BNEQ:	if (A != C) then PC = ALUout	
3	STALL		

Instruction Load upper / lower Imm.	**Type** **I**	**Op** **0110 / 0111**	**func** **-**
Cycle	**Label**	**RTL**	
2	LUI:	R1 = IMM8[8-0] << 8	
	LLI:	R1 = ZE(IMM8[8-0])	

Instruction Load to register	**Type** **I**	**Op** **1000**	**func** **-**
Cycle	**Label**	**RTL**	
2	LTR16:	if IMM16 then I0 = IMM16[15-8] << 8 I1 = ZE(IMM16[7-0])	
	LTR8:	if IMM8 then R1 = SE(IMM8)	
3	LTR16:	ALUout = I0 | I1	
4	LTR16:	R1 = ALUout	

A1. Design Documentation

Instruction Copy to register	**Type** **I**	**Op** **1001**	**func** **-**
Cycle	**Label**	**RTL**	
2	CTR:	R1 = C	

Instruction Load word	**Type** **I**	**Op** **1010**	**func** **-**
Cycle	**Label**	**RTL**	
2	LD FROM MEM:	MDR = Mem[C]	
3	LWSTALL		
4	LOAD TO REG:	R1 = MDR	
5	STALL		

Instruction Store word	**Type** **I**	**Op** **1011**	**func** **-**
Cycle	**Label**	**RTL**	
2	SW:	Mem[C] = A	
3	STALL		

A1. Design Documentation

Instruction syscall	**Type** **I**	**Op** **1110**	**func** **-**
Cycle	**Label**	**RTL**	
2	INCR PC:	PC = PC + 1	
?	READ:	if FINISHED then goto RDONE	
	WRITE:	if FINISHED then goto WDONE	
	EXIT:	Close program	
?	RLOOP:	if FINISHED then goto RDONE else goto RLOOP	
	WLOOP:	if FINISHED then goto WDONE else goto WLOOP	
?	RDONE: / WDONE:		

Instruction Jump Register	**Type** **I**	**Op** **0101**	**func** **-**
Cycle	**Label**	**RTL**	
2	JR:	PC = A	
3	STALL		

Instruction -	**Type** **L**	**Op** **0011 / 0100**	**func** **-**
Cycle	**Label**	**RTL**	
2	J:	PC = (IMM12) | PC[15-12]	
	JAL:	RA = PC PC = (IMM12) | PC[15-12]	
3	STALL		

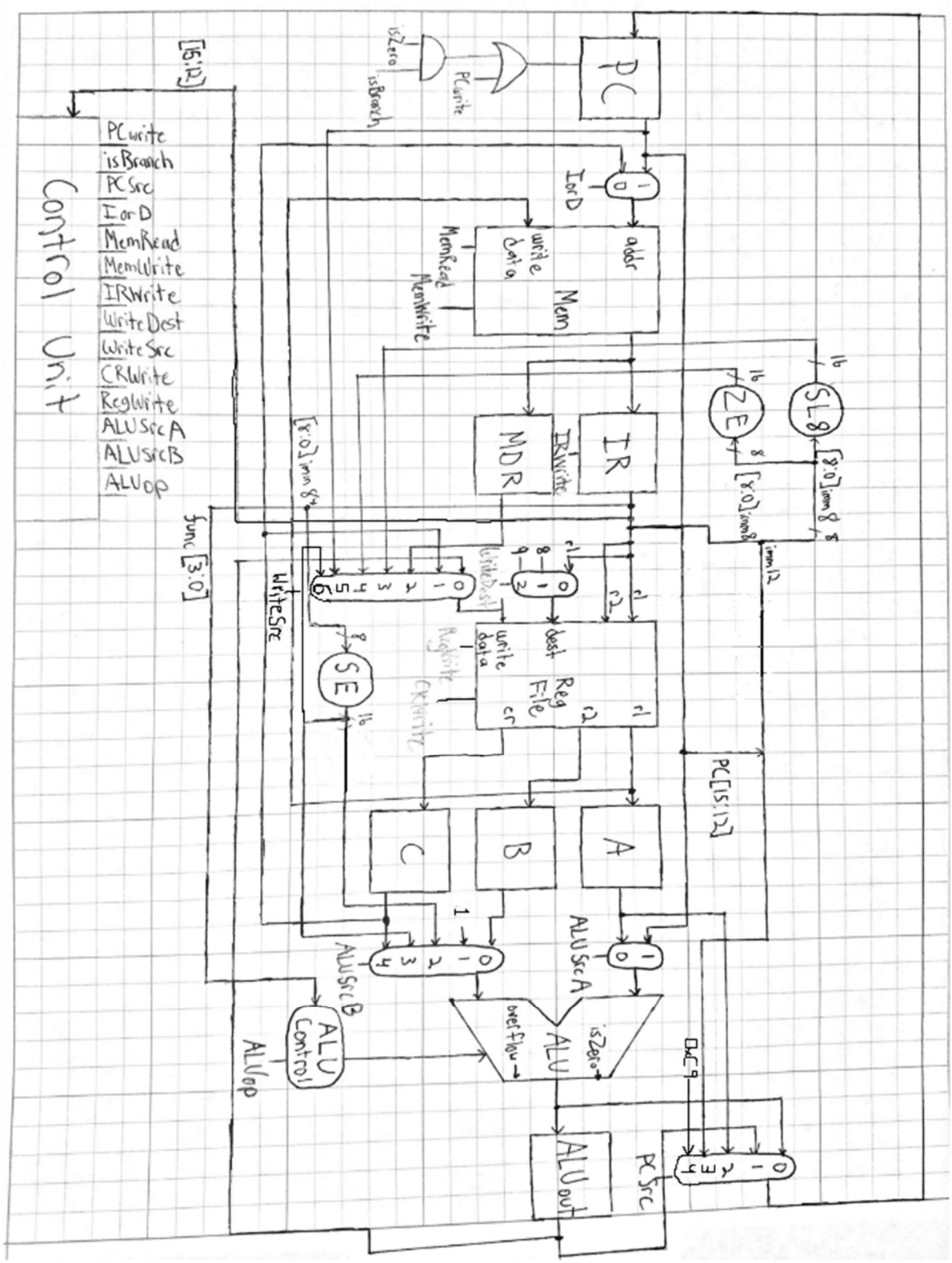
Datapath Components

Table of Components and Signals

Signals Components	Inputs (size)	Outputs (size)	Description of Component
PC	- newPC 16'b	- PC 16'b	- PC => Program counter
Memory	- Address 16'b - Write Data 16'b	- MemData 16'b	- Access to memory
IR	- Instruction Data 16'b	- r1 4'b - r2 4'b - imm 8'b or 12'b	- IR => instruction register - Stores instruction data
MDR	- Memory Data 16'b	- Memory Data 16'b	- MDR => Memory data register - Stores memory data
Register File	- r1 4'b - r2 4'b - WriteDest 4'b - Write Data 16'b	- r1 data 16'b - r2 data 16'b	- Registers that store data for reading/writing
A/B Registers	- A or B respectively 16'b	- A or B respectively 16'b	- These registers hold operands for the ALU
ALU	- A 16'b - B 16'b	- isZero 1'b - Overflow 1'b - Operation Result 16'b	- Performs operation on two operands
Shift Left by 1	- Imm 12'b	- Shifted imm size + 1	- Shifts immediate by 1
Shift Left by 8	- Imm 8'b	- Shifted imm size + 8	- Shifts immediate by 8
Sign Extender	- Imm 8'b	- Sign Extended imm 16'b	- Fill in top bits of immediate with sign of immediate
Zero Extender	- Imm 8'b	- Zero extended imm 16'b	- Fill in top bits of immediate with 0s

A1. Design Documentation

Block Diagram of Datapath



Control Unit

Table of Control Signals

Control Signal	Size	Component	Description
PCSrc	3'b	PC Source (Mux)	Controls the source of PC: incremented PC, jump address, or branch address.
PCWrite	1'b	PC	Controls whether PC should be written to
isZero	1'b	PC	ALU isZero result *Branch condition
isBranch	1'b	PC	Controls whether an instruction is a branch *Branch condition
isBIEQ	1'b	ALU	Determines whether an branch instructions is a bieq
IorD	1'b	MemAddr (Mux)	Decides whether a memory address is coming from PC (instruction) or the C register (data)
MemRead	1'b	Memory	Controls whether memory is being read
MemWrite	1'b	Memory	Controls whether memory is being written to
IRWrite	1'b	IR	Controls whether IR is being written to
WriteDest	2'b	RegDest (Mux)	Controls whether destination of a register write is $cr or r1
WriteSrc	3'b	WriteData (Mux)	Controls whether source of data to be written to a register (MDR, ALout, C, etc.)
CRWrite	1'b	Register File	Controls whether the $cr register can be written to (ie disable $cr being set as r1 in **I-type** instructions)
RegWrite	1'b	Register File	Controls whether a register is being written to
ALUSrcA	1'b	ALUSrcA (Mux)	Controls whether the source of ALU input A is PC or the A register
ALUSrcB	2'b	ALUSrcB (Mux)	Controls whether the source of ALU input B is one of three immediates or the B register
ALUop	2'b	ALUControl	Controls the operation sent to the ALUControl component

A1. Design Documentation

Specification for Controls:

PCSrc
- 000 - PC + 1
- 001 - ALUout
- 010 - Register A
- 011 - (PC[15-12] | IMM12)
- 100 - Addr. of Exception Handler

PCWrite
- 0 - turns off; 1 - turns on

isBranch
- 0 - not branch
- 1 - is branch

isBIEQ
- 0 - is bieq instruction
- 1 - is not bieq instruction

IorD
- 0 - C register (data)
- 1 - PC (instruction)

MemRead
- 0 - turns off; 1 - turns on

MemWrite
- 0 - turns off; 1 - turns on

IRWrite
- 0 - turns off; 1 - turns on

WriteDest
- 00 - r1
- 01 - $cr
- 10 - $ra

WriteSrc
- 000 - ALUout
- 001 - Register C
- 010 - MDR
- 011 - (IMM8 << 8)
- 100 - ZE(IMM8)
- 101 - PC
- 110 - SE(IMM8)

CRWrite
- 0 - turns off; 1 - turns on

RegWrite
- 0 - turns off; 1 - turns on

ALUSrcA
- 0 - Register A
- 1 - PC

ALUSrcB
- 00 - Register B
- 01 - 1
- 10 - SE(IMM8)
- 11 - Register C

ALUop
- 00 - Add (PC increment)
- 01 - Sub (Branch)
- 10 - function code (C-type)
- 11 - OR

func
- 4’b function determined by C-type

opcode
- Determined by instruction

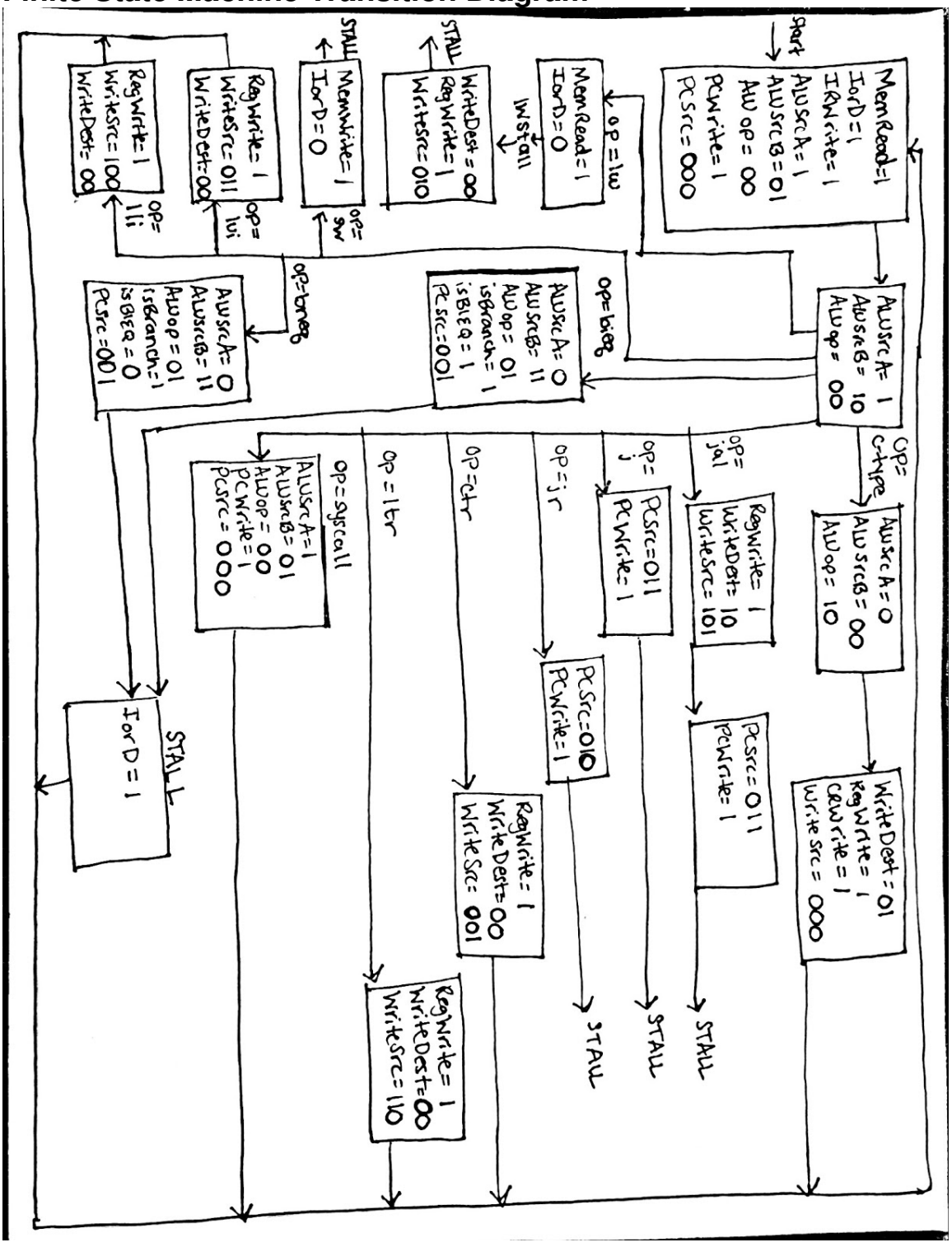
current_state/next_state
- Keeps track of state in multi-cycle

CLK
- Clock speed

Reset
- 0 - Don’t Reset; 1 - Reset

A1. Design Documentation

Finite State Machine Transition Diagram



Implementation and Testing

Hardware Implementation Plan:

Special Registers (PC, IR, MDR, A, B, C):

- Each register will be able to hold 16-bits and have one output with as much.
- They will have multiple inputs such as the clock signal, a reset signal, and an input for the data.
- These will be made with a Verilog module in Xilinx.

Memory:

- For each bit, use a flip flop.
- Use buses and muxes to combine the data.
- Built as a schematic using these parts in Xilinx.

Register File:

- The register file will be made of registers and parts for the logic required to choose the correct outputs.
- 4'b address input and value output
- The cr register will always be outputted to a special C register.
- The register file will be modified from the resource provided to us to handle 16 registers.

ALU:

- For a 16-bit ALU, we will add a unit for Arithmetic (adds and subs), a unit for Logic (OR, NOR, XOR, AND, NAND), and a unit for less than (slt).
- Each unit will have a 16-bit input and output with their outputs connected to a mux.
- The mux chooses which data to output of the ALU, based on the ALU control signal.
- The ALU can be built in Xilinx using schematics, a modified design from the 4-bit ALU built in Lab 6.

Sign Extender:

- A simple verilog module that takes the most significant bit of the input and extends it to the left, extending the size of the data to 16 bits.

Zero Extender:

- Another verilog module that fills top bits with zeros, extending the size of the data to 16 bits.

Shifters:

- Verilog module that will take input and shift the bits to certain amount.

Muxes (1'b, 2'b, 3'b, 4'b)

- Modify resources given to us

Unit Testing:

General Testing:

- Consider instruction description including inputs and outputs
- Draw out the path of data flow from one component to another
- Confirm desired data source input and output for each component
    - Upon error return to last step and debug

    - Repeat until problem is found
- Confirm desired result/action at the end of calculation

Regular and Special Registers (A, B, C, IR, MDR):

Overall, all of these registers have one thing that they need to accomplish. They need to be able to take in values, hold them and then output that same value on the output for that register.

Memory:

The memory needs to able to hold the instructions that we will be using for all operations and other values that the user might want to store into it after the processor starts doing operations. The memory should be able to output any instruction we have stored in it as well as any data when called upon.

Register File:

The register file needs to be able to select any registers located within it (Note: The only situation that it should not be able to select the proper register is if the user tries selecting a register that they are not allowed to manipulate themselves). These registers, just like the previous ones, must be able to hold any and all data values presented to them and output them on their respective output. It must be able to write any data to the proper register when called upon (same restrictions as note).

ALU:

The ALU is a combination of different logic gates that can perform different tasks, arithmetic and logical. At the end, it needs to be able to select the correct output among the multiple outputs it will calculate. Below are all the possible operations we will need to test. There are also two flags that we will need to make sure they are set at proper times, they will be tested with the functions that will need to be able to manipulate these flags.

- Addition
    - Needs to be able to handle big additions that cause overflow like numbers out of the range of a 16 bit data type.
    - Needs to be able to handle adding positive and negative numbers, those that will and will not cause overflow. Basically, detect overflow.
- Subtraction
    - Be able to set an IsZero flag upon subtracting two values to get zero.
    - Needs to be able to handle subtracting positive and negative numbers from each other, those that will and will not cause overflow.
    - Needs to be able to set a Negative flag to represent the confirmation that first number in the argument is less than the second number in the argument.
- Bit AND any two numbers of similar bit size
- Bit OR any two numbers of similar bit size
- Bit NOR any two numbers of similar bit size
- Bit XOR any two numbers of similar bit size
- Bit NAND any two numbers of similar bit size

Control Unit:

The control unit must be able to take in an instructions opcode and proceed to handle the flow of cycles. This means that for each instruction, the transition of the datapath's

state must be checked at every stage to ensure correct control bits for the respective cycle.
This will likely need to be tested in Verilog with state transitions and simulations for each instruction.

ALUControl:

This will be tested to ensure that in can take in the function bits decoded from the IR register and hold them until the control finishes decides what it needs to do. It should then be able to either send the function code to the ALU if it needs to use it or send a different function code than what was originally sent into the control.

Zero/Sign Extenders:

- Must be able to extend any immediate we give correctly.

Shifters:

- Shift Left 1:
  - Must be able to shift in a zero to the right of any immediate we give it, even a 16 bit immediate (the bit on the far left would be lost after the shift in this situation).
- Shift Left 8:
  - Must be able to shift in a zero to the right, like the Shift Left 1, just seven more times.

Integration plan and testing:

General Note: Assume all steps are independent of each other and only require listed components.

Step 0:

Goal: Increment PC
Components: PC, ALU (No Mux)
Testing:
General:
- Input A: PC
- Input B: 1
- Op: XXXX (Add)
- Output: PC + 1

Edge Cases:
- N/A

Step 1:

Goal: Read Memory
Components: PC, MemAddr Mux, Memory, MDR (Read-only)
Testing:
General:
NOTE: PCWrite is set to 1
- MemAddr Mux properly selects 1
- Correct address is returned from MemAddrMux
- With MemRead set to 1, value at PC's memory address is outputted
  - MDR value is now value at PC's memory address
- With MemRead set to 0, no value is outputted

- MDR value is unchanged

Edge Cases:

- Memory address doesn't exist
- No data in PC
- No data at that memory address
- MemWrite set to 1, behavior is expected

Step 2:

Goal: Write Memory
Components: MemAddr Mux, Memory (Write only)
Testing:

General:

- MemAddr Mux properly selects 0
- Correct address is returned from MemAddrMux
- With MemWrite set to 1, value at address is set to write data
- With MemWrite set to 0, value at address is unchanged

Edge Cases:

- With MemRead set to 1, behavior is expected
- Memory address doesn't exist

Step 3:

Goal: Read Register File motivated by IR
Components: IR, Register File (Read-only)
Testing:

General:

- When IRWrite is set to 1, IR is overwritten
- When IRWrite is set to 0, IR is unchanged
- Bits 11-8 from the Instruction Register go to r1 in the register file
- Bits 7-4 from the Instruction Register go to r2 in the register file
- The value in r1 is correctly outputted from the register file
- The value in r2 is correctly outputted from the register file
- Value in $cr is outputted

Edge Cases:

- Bits 11-8 do not refer to an existing register.
- Bits 7-4 do not refer to an existing register.
- The Instruction Register has no data.
- IRWrite is set to 0, and no data is in IR

Step 4:

Goal: Write to Register File
Components: WriteDest Mux, WriteData Mux, Register File (Write only)
Testing:

General:

- WriteDest Mux can select between 2 - $ra, 1 - $cr and 0 - another register and output correct value
- WriteData Mux can select between 5 sources and output correct value
- When RegWrite is set to 1 and WriteDest is set to 0, correct register stores correct value after write

- When RegWrite is set to 0 and WriteDest is set to 0, register destination is unchanged
- When CRWrite is set to 1, RegWrite is set to 1, and WriteDest is set to 1, $cr contains
- When CRWrite or RegWrite is set to 0, and WriteDest is set to 1, $cr is unchanged
- No other register should be written to

Edge Cases:

- WriteDest does not exist (input of >= 3)
- WriteData does not exist (input of >= 5)

Step 5:

Goal: ALU ops and outputs

Components: ALUSrcA Mux, ALUSrcB Mux, ALU, ALUControl

Testing:

General:

- ALUSrcA Mux can select between 2 sources and output correct value
- ALUSrcB Mux can select between 4 sources and output correct value
- ALUControl outputs proper value based on ALUop and func
- When computation results in zero, isZero output is 1, otherwise output is 0
- When computation results in overflow, overflow output is 1, otherwise 0

Edge Cases:

- Func code does not exist
- Input for logical operations are different in bitsize
- Overflow occurs
- ALUop requires function code as output

Step 6:

Goal: PCSrc and control signals

Components: PCSrc Mux, PCWrite, isBranch, isZero, PC

Testing:

General:

- PCSrc properly selects between 4 sources and outputs correct value
- PCWrite set to 1, PC is changed to PCSrc output
- PCWrite set to 0, isBranch and isZero set to 1, PC is changed to PCSrc output
- PCWrite set to 0, isBranch or isZero set to 0, PC is unchanged

Edge Cases:

Step 7:

Goal: Shifters and extenders

Components: Shifters, Extenders

Testing:

General:

- Shift Left 1:
  - Ensure any sized input is shifted left one and outputted with original size + 1
- Shift Left 8:
  - Ensure any size input is shifted left eight and outputted with original size + 8
- Zero Extend:
  - 8-bit input is outputted with zero extension
- Sign Extend:
  - 8-bit input is outputted with sign extension

Edge Cases:

- Data of incorrect size is inputted

Final Step:

Combine pieces of processor together and proceed to system testing.

System test plan:

a. Individual instructions
b. Small programs
   i. Code Fragments
   ii. Small loop
   iii. Small programs
a. Simple arithmetic and logic
b. Simple jumps
c. Large programs
   i) relPrime (see pg. 13-20)

Deisgn Process Journal

Meeting 1: Tuesday, 10 January 2017

We began the meeting by discussing the features we most wanted to include in our processor. We hoped this would guide our design throughout the meeting.

We decided to build a processor that is primarily based on a combination of load-store and accumulator designs. One specific register will hold the value of the last arithmetic or logical computation. This register will be read only for users to copy and read the value of the last computation. We will also allow users a small set of registers to save values beyond a computation and save values over procedure calls.

Our processor will have 18 registers in total, 15 of which are available to the user in some capacity (read/write). There are 8 general purpose registers, and 8 special registers. We decided on a small, specific instruction set made up of one major type, an arithmetic/logical type that returns all results to the special computation register, as well as other instructions of varying sizes. We believe that varying sizes will help keep our programs small in size and more efficient.

For procedure calls, we thought it would be interesting to make arguments and return values memory addresses only. We acknowledge the inefficiencies of that design but felt that the value of having direct access to the memory address of the arguments and return values was a valuable asset to our design.

Work log:
(majorly a group effort worked on during the 3 hour period):
Design and description of registers, instruction type and format, procedure call conventions (Discussion)
Trinity - Journal Notes

Meeting 2: Wednesday, 11 January 2017

We are doing a lot of redesigning based on feedback and more direction with the project. We decided to standardize the size of instructions to 16 bits. We now have two types of instructions, a C-type for register to register computations and an I-type for other instructions that require a register and immediate values such as load/store, branch, jump. The I-type include all of our instructions with previously varying sizes. By establishing a standard size and design for instructions, we are able to greatly simplify our design and get a better direction on designing instructions.

A2. Design Process Journal

We decided to scrap the memory address-only idea for arguments and return values as it would be a cumbersome design at our current state of progress.

By the end of the day, we've ended up with 20 registers, deciding to split our general purpose registers between saved and temporary registers. Our number of instructions has grown to include a number of pseudoinstructions as Shaun began coding the programs.

Work log:
Before Meeting:
Khaled and Shaun - Register Descriptions (write up)
Over 7 hour group time (1800-0100):
Logan - Assembly Syntax and Semantics, addressing modes, grammar and formatting
Trinity - Procedure Call Conventions (write up), Journal, Machine language instruction format type and semantics, rules for translating assembly to machine language
Khaled - Assembly fragments
Shaun (assisted by Trinity) - Euclid's algorithm and relPrime (Assembly)
Group - moderate redesign of instruction format, registers, and instructions
After meeting (2 hours):
Shaun (assisted by Khaled) - Euclid's algo/relPrime (Machine Language)

Meeting 3: Thursday, 12 January 2017
We decided to reformat the jal and j instructions to be their own type, L-type. This allowed for a larger jump block for us,

We plan on featuring a assembler, compile, linker, and exception handler at the very least in our processor. One of the registers dedicated to exception handling will be $ex, the cause register. We also cut down our registers to make our register file size 16.

We created a few new instructions and pseudoinstructions and added these to the design documentation. Most important of these is probably the addition of *syscall* which will be our I/O instruction.

Other than these decisions, we reformatted and reworded our design document according to the feedback provided to us and attempted to make things more clear.

Work log:
Over 1 hour meeting:
Logan – general editing over entire doc
Trinity – syscall, editing instruction descriptions
Shaun – fixed problems with Machine Language
Group – redued register file and redesigned jal/j instructions.

Meeting 4: Friday, 13 January 2017
We polished off the work from yesterday including updating our common operations table, reviewing the changes made yesterday, and finishing any work left incomplete after the meeting.

Began going over Milestone 2 requirements and planning for the next few days.

Work log:
Over 1 hour meeting:
Shaun – Machine code and common operations table
Trinity – finished syscall, journal
Group – Formatting and decisions about new instructions

Meeting 5: Sunday, 15 January 2017
We decided to use multi-cycle control as we thought this would make our performance faster and simplify our RTL design. We found that we could group all C-type and L-type instructions into individual RTL, and grouped branches, load/store word, and load lower/upper immediate instructions as well.

Planned meeting times:
Sunday: 1400-1700 (may vary)
Mon-Fri: 2030-2130, class time and lab time when cancelled

Work log:
Over 3 hour meeting:
Shaun – RTL Table, Component test descriptions
Logan – RTL Table, Component list,
Trinity – RTL Table, journal, formatting, editing
Khaled – Control Descriptions, input/output/control signals
Group – Discussion on major decisions

Meeting 6: Monday, 16 January 2017
No major decisions.

Worked on milestone 3.

Work log:
Over 1 hour meeting:
Trinity - Journal
Khaled – worked on Milestone 2
Shaun – Reworked Machine Language translations to conform to changes
Logan/Trinity (w/ assistance from Khaled & Shaun) – Datapath Design

Meeting 7: Tuesday, 17 January 2017
No major decisions.

Polished milestone 2.

Work log:
Over 1.5 hour meeting:
Logan – Inputs and outputs, RTL modifications
Logan and Shaun – Test descriptions
Trinity – RTL/control description/input & output modifications, additional components, journal
Logan, Shaun, & Trinity – General editing, formatting and corrections
Khaled – sick (not in attendance)
After meeting (~1 hour):
Trinity – Reviewed/modified RTL and Verilog to adhere to test descriptions and page numbers!!

Meeting 8: Thursday, 19 January 2017
Worked on Lab 7 & 8

Work log:
Over 1 hour meeting:
Shaun and Khaled – Lab 7
Logan and Trinity – Lab 8

Meeting 9: Friday, 20 January 2017
Reviewed feedback for milestone 2 and made some major changes to RTL and organization of design document.

Work log:
Over 1 hour meeting:
Khaled – broke up RTL tables, edited RTL, formatting (tables, color)
Logan and Trinity – inputs/outputs table, table formatting, review of changes
Trinity – Condensed/reordered machine language and assembly sections, journal
Shaun – Reviewed tests, reviewed changes, small editing

Meeting 10: Sunday, 22 January 2017
Worked on Lab 7 & 8

Work log:
Over 3 hour meeting:
Shaun and Khaled – Lab 7
Logan and Trinity – Lab 8
Trinity - Journal

Meeting 11: Monday, 23 January 2017
Worked on Lab 7 & 8

Work log:
Over 1 hour meeting:
Shaun and Khaled – Lab 7 and RTL review
Logan and Trinity – Lab 8
Trinity - Journal

Meeting 12: Tuesday, 24 January 2017
In our RTL for the instructions *load upper immediate, load lower immediate* and *load to register*, we decided to denote the immediate value by the bits in the 16-bit immediate itself rather than where they are in the instruction. We weren't sure how to access the correct parts of the instruction. In these instructions, we changed some logical code to simply shift the immediates left by 1.

We also decided to add a C register component that receives the value of $cr no matter what input goes into the register file.

Work log:
Over 2 hour meeting:
Trinity and Logan (with assistance from Shaun and Khaled) – Revised and finished datapath
Trinity - Journal
Everyone – Revised RTL and divvied up assignment
After meeting:
Logan – Block diagram of datapath (estimated 1 hours) **Actual: expected**
Implementation plan for each component (estimated 1 hour) **Actual: 20 minutes**
Khaled – Implementation and tests of Mem, ZE, and SE (estimated 2 hours) **Actual: Unknown/Incomplete**
Shaun – Update unit tests (estimated 30 minutes) **Actual: 5 minutes**
Integration plan and testing (estimated 1 hour) **Actual: 20 minutes**
Trinity – Control signal descriptions (estimated 30 minutes) **Actual: expected**
Specs for Control Unit in Component List (estimated 1 hour) **Actual: expected**
Implementation and tests of ALU, SL1 (estimated 4 hours) **Actual: incomplete**

DISCUSSION OF TEST STRATEGY

Our testing strategy consisted mostly of trying to anticipate problems before they occurred. We think of what might go wrong with a component and write tests to catch those situations. After that, we write general cases for expected outputs with non-extreme or expected data. When these tests are finished, we run through all inputs that satisfy the tests and make sure the output is expected/acceptable.

HOW OUR ARCHITECTURE CHOICE AFFECTED OUR DATAPATH

Our choice in architecture – Load/Store with Accumulator, caused us to have a lot of extra muxes to choose the value from our special computation result register as well as a special register in the datapath that always stores the value of the computation result register. Al of this was caused by our accumulator computation register.

Because our design is also load/store, we found it necessary to have a memory unit as well as a register file to hold data values. Between these two components and our ALU, we found it necessary to have special temporary register such as IR, MDR, A/B/C to hold necessary values between stages in the datapath.

Meeting 13: Thursday, 26 January 2017
Reviewed feedback and made changes to document.

Work log:
Over 1.5 hour meeting:
Trinity and Logan – Revised RTL, planning/revising datapath, integration plan
Trinity – Reorganized document based on feedback, journal
Shaun – Unit Tests, integration plan
Khaled – Table colors, reviewed milestone 4, minor edits and changes
After meeting:
Logan – Making implementation plan more robust (estimated: 30 minutes)
Datapath redesign (estimated: 1 hour)
Trinity - Integration plan (estimated: 1- 2 days) **Actual: 1.5 hours**

Meeting 14: Friday, 27 January 2017
Reviewed feedback and made changes to document.

Work log:
Over 1.5 hour meeting:
Logan (with assistance from Trinity) – Rough draft of state transition diagram, went over datapath corrections
Trinity – Editting/formatting design document, journal
Shaun and Khaled – not in attendance
After meeting:
Logan – Making implementation plan more robust (estimated: 30 minutes)
Datapath redesign + control unit (estimated: 2 hour)
Trinity – Official finite state machine transition diagram (2 hours)

Meeting 15: Monday, 30 January 2017
Changes to control and datapath to be in accordance with RTL (controls affected: WriteSrc and ALUSrcB).

Work log:
Before meeting:
Shaun – Shift left 8 **(approx. 1 hour)**
Khaled – Shift left 1 **(approx. 1 hour)**
Over 1 hour meeting (and after meeting):
Trinity – Register file and test bench **(approx. 6 hours)**, control unit test descriptions **(estimated 1 hour)**, journal
Khaled – Muxes **(estimated 1 day)**
Logan – datapath redesign (again – sorry, Logan) **(approx. 1 hour)**, ALUControl **(estimated 2 hours)**
Shaun – not in attendance

Meeting 16: Tuesday, 31 January 2017
No major changes.

Work log:
Over 1 hour meeting:
Trinity – Register File, journal
Logan – ALUControl **(estimated 2 hours)**
Shaun – test benches for unit tests
Khaled – not in attendance
After meeting:
Trinity – Register file and test bench **(approx. 6 hours)**, control unit test descriptions **(estimated 1 hour)**, start integration plan and testing

Meeting 17: Wednesday, 1 February 2017
No major changes

Work log:
Before meeting:
Trinity – Register File changes, ALUControl and test bench **(approx. 3 hours)**, journal
Khaled and Shaun – ALU16b & test bench **(approx. 2 hours)**
Over 1 hour meeting:
Trinity – Control unit **(plus 30 min)**, journal
Khaled – ALU16b
Logan – began integration plans
Shaun – not in attendance

Meeting 18: Thursday, 2 February 2017
No major changes.

Work log:
Before meeting:
Trinity – Control Unit **(approx. 1 hour)**
During 6th period:
Shaun – test benches, and modifying/correcting muxes
Trinity – Control unit test bench, journal

Meeting 19: Friday, 3 February 2017
No major changes.

Divvied up integration plan.

Work log:
Over 1.5 hour meeting:
Logan – Incrementing PC (PC, ALU), Write/Read to/from memory
- Add revised datapath to design doc

Khaled – Write/Read to/from Register File w/ IR
Shaun – ALU and ALU Control
Trinity – PCSrc, shifters, extenders
- Rewrite Transition Diagram **(approx. 1 hour)**
- Control Unit implementation and test **(approx. 2 hours)**
- Journal

Meeting 20: Sunday, 5 February 2017
No major changes

Work log:
Before meeting:
Logan - Incrementing PC (PC, ALU), Write/Read to/from memory **(approx. 2 hours)**
Over 1.75 hour meeting:
Khaled – Write/Read to Register File w/ IR
Shaun – ALU and ALUControl **(approx. ½-1 hour after meeting)**
Logan - Incrementing PC (PC, ALU), Write/Read to/from memory
-Schematic done, errors causing inability to test

Trinity – PCSrc **(approx. 1.5 hours after meeting)**
- Journal

Meeting 21: Monday, 6 February 2017
No major changes

Over 1 hour meeting:
Khaled – researched assemblers
Trinity – building datapath, journal
Logan – memory fixes
Shaun – not in attendance
After meeting:
Trinity – building datapath **(approx. 3 hours)**

Meeting 22: Tuesday, 7 February 2017
PC + 2 changed to PC + 1.

Divvied up extra features:
*Program tools – Khaled and Shaun
*Exceptions and Interrupts – Logan
*I/O – Trinity

Work log:
Over 2 hour meeting:
Khaled – Assembler research
Logan – memory fixes
Shaun – preparation for system tests
Trinity – building datapath, journal
After meeting:
Shaun – system tests **(estimated: unknown)**

Meeting 23: Wednesday, 8 February 2017
No major changes

Work log:
Over 1.5 hour meeting:
Trinity and Shaun – system testing and reworking
Trinity - Journal
Khaled – Assembler
Logan – Exception Handler
After meeting:
Trinity – system analysis **(approx. 3 hours)**

Meeting 24: Thursday, 9 February 2017
No major changes

Work log:
Over 1 hour meeting:
Logan – Exception handler
Khaled – Assembler
Shaun – robust system tests
Trinity – Modifications to data path to initialize $st, journal

Meeting 25: Friday, 10 February 2017
No major changes

Work log:
Over 2 hour meeting:
Logan – Exception handler (first hour)
Khaled – Assembler
Shaun – relPrime coe
Trinity – instructions coe, journal
After meeting:
Trinity – debugging processor (where it pertains to branches and jump intstructions)

Meeting 26: Sunday, 12 February 2017
Fixing processor to handle PC + 1:

- Branch is directly translated -> PC = SE(BranchAddr)
- Jumps are directly translated -> PC = PC[15:12] | IMM12
- Stalls added to all branch and jump instructions

Work log:
Over 3 hour meeting:
Logan – Exception handler
Shaun – looked into compiler (not going to work on this), started IO
Khaled – Assembler
Trinity – debugging processor, journal

Meeting 27: Monday, 13 February 2017
Major changes:

- Stalls for sw and lw
- STALL control bit: IorD = 1
- Separate controls for bieq/bneq
    - isBIEQ added on ALU

Work log:
Before meeting:
Trinity – system analysis and fixing relPrime **(approx. 3 hours)**
Over 2 hour meeting:
Trinity – making relPrime work, journal
Logan, Khaled, Shaun – not in attendance
After meeting:
Trinity – worked on datapath to incorporate changes, redrawing state transition diagram **(approx. 2 hours on 15 Feb 2017)**

- reworking relPrime to actually function properly and changing assembly/machine code in design doc **(approx. 4 hours on 15 Feb 2017)**
- Worked on system to make it work - added stall between lw1 and lw2 **(approx. 2 hours on 16 Feb 2017)**
- More fixing to actual working **(approx. 3.5 hours on 17 Feb 2017)**
- Began implementing I/O **(approx. 1.5 hours 17 Feb 2017)**
- Redrew state transition diagram **(approx. .5 hours)**

Meeting 28: Sunday, 19 February 2017
Overhaul of memory and small changes to accomodate (using wrong type of block memory for the FPGA board).

Work log:
Over 2.5 hour meeting:
Trinity – Report, M6, fixing memory and other changes, journal
Shaun – IO
Logan – Presentation
Khaled - Assembler

Meeting 29: Tuesday, 21 February 2017
PC starts at 76

Work log:
Over 3 hour meeting:
Trinity – IO, performance data, journal
Logan – Final Report

Test Results

1a.	**Total Bytes Required for Euclid's Algorithm**	200B
1b.	**Total Bytes Required for relPrime Algorithm**	614B
2.	**Total Instructions executed;** ***relPrime, n = 0x13B0***	36,360 instructions
3.	**Total Cycles;** ***relPrime, n = 0x13B0***	40,099 cycles
4.	**Average Cycles per Instruction** (based on 2 and 3)	1.09 cycles/instruction
5.	**Cycle Time for Design** (from Xilinx Synthesis Report – Timing Summary)	17.114 ns =58.431 Mhz
6.	**Total Execution Time;** ***relPrime, n = 0x13B0***	7,355,580ns = 7.35558ms
7.	**Gate Count for Design**	-
8.	**Device Utilization Report** (from Xilinx Map Report)	

Number of errors:	0
Number of warnings:	7
Logic Utilization:	
Total Number Slice Registers:	441 out of 9,312 4%
Number used as Flip Flops:	405
Number used as Latches:	36
Number of 4 input LUTs:	788 out of 9,312 8%
Logic Distribution:	
Number of occupied Slices:	575 out of 4,656 12%
Number of Slices containing only related logic:	575 out of 575 100%
Number of Slices containing unrelated logic:	0 out of 575 0%
**See NOTES below for an explanation of the effects of unrelated logic.*	
Total Number of 4 input LUTs:	835 out of 9,312 8%
Number used as logic:	788
Number used as a route-thru:	47

A3. Test Results

The Slice Logic Distribution report is not meaningful if the design is over-mapped for a non-slice resource or if Placement fails.

Number of bonded IOBs:	91 out of 232 39%
IOB Flip Flops:	16
Number of RAMB16s:	1 out of 20 5%
Number of BUFGMUXs:	1 out of 24 4%
Average Fanout of Non-Clock Nets:	3.18
Peak Memory Usage:	343MB
Total REAL time to MAP completion:	6 secs
Total CPU time to MAP completion:	3 secs

NOTES:

Related logic is defined as being logic that shares connectivity - e.g. two LUTs are "related" if they share common inputs. When assembling slices, Map gives priority to combine logic that is related. Doing so results in the best timing performance.

Unrelated logic shares no connectivity. Map will only begin packing unrelated logic into a slice once 99% of the slices are occupied through related logic packing.

Note that once logic distribution reaches the 99% level through related logic packing, this does not mean the device is completely utilized. Unrelated logic packing will then begin, continuing until all usable LUTs and FFs are occupied. Depending on your timing budget, increased levels of unrelated logic packing may adversely affect the overall timing performance of your design.