Imperial College London

ELEC40006-Electronics Design Project 1 2019-2020

CPU Design – SEGFAULT

Report for

Dr. Edward Stott and Mrs. Esther Perea

Alexander Pondaven, CID 01730117, Undergraduate Electrical and electronic engineering

Shuanghua Liu, CID 01729607, Undergraduate Electrical and electronic engineering

Yuliang Zhu, CID 01771679 Undergraduate Electrical and electronic engineering

Word count: 8305

Submitted June 2020

Contents

1 Abstract

This report presents in detail how a Central Processing Unit has been built to meet certain benchmarks and design goals. The CPU was based on an invented instruction set architecture and took advantage of the Harvard Architecture philosophy, while building off the work done with ARMish in term two. In this report, a clear explanation of the hardware implementation has been done including analysis of how individual instructions control hardware. The report also includes the optimisation process, where pipelining, latency/propagation delay and the trade-off between speed, power, and area have been discussed.

2 Introduction

The CPU is the core of a computer and plays a paramount role in electronic engineering. The primary objective was to build a CPU that could do the benchmarks (Calculating Fibonacci numbers, pseudo-random integers with a linear congruential generator and traverse a linked list), while remaining general (Turing complete). Hence research on different ISAs like AVR, MIPS, SPARC, and ARM, as well as the Harvard Architecture was conducted. Some instructions were specifically developed for the benchmarks like multiply, subroutines, and stacks. After deciding upon a 12 instruction ISA to meet the benchmark requirements, the separate hardware blocks were decided. The CPU's hardware took inspiration from the ARMish CPU built in term two, as it had a register file and ALU to carry out calculations. After implementing all instructions with hardware, the CPU was optimised to maximise power, speed, and area. This was done through pipelining, parallel computation, and analysis of different multipliers.

3 Project Planning and Management

Before starting the project, the Belbin Inventory had been completed to find out the different roles of team members. There was an implementer who is practical, reliable, efficient, hard-working, and methodical. It helped to have a shaper who is energetic, driven, and bold. The final person was a resource investigator who is outgoing and a great motivator. The task was split into three main tasks for each member, with all team members having a general understanding of the CPU. One person implemented the hardware and tested instructions.

The second person had done research on the multiplier and hardware components. The last person helped write the test code for the benchmarks and implement the CPU. On 8th June, all the general instructions with associated hardware had been completed and tested. On 10th June, all three benchmarks had been completed and tested. On 12th June, the CPU had been optimised to achieve a better frequency. A regular meeting pattern had been used. All team members met every day since 24th May at 10:00 BST on Microsoft Teams to discuss progress and distribute tasks for the day. The TODO list website, Trello, and One Note were used to make share notes throughout the project. Some screen shots of the meeting plans and to do lists are in Appendix 3.

Snippet of team notes:

Date	Discussion	Alex	Peter	Jason
26/6	• Discussed Jason's assembly code for fib benchmark • Went over stacks • Discussed number of registers and instruction format ○ Decided it is best to decide this once all benchmarks have been turned into assembly and we know what instructions will be needed	Look at next benchmarks and see what instructions we may need and convert to assembly	Research multiplication methods	Make assembly more efficient and think of register to use
27/5	• Went over assembly code for LCG and linked list benchmarks • Discussed multiplication methods • Made a list of instructions using benchmarks	Think of implementation of instructions (DECA oral as well)	Continue multiplication method research	Reduce number of instructions used in fib

4 Design Criteria

The CPU should be able to run the following three benchmarks.

Calculate Fibonacci numbers using recursion

```
int fib(const int n){
     int y;
     if (n <= 1) y = 1;
     else {
         y = fib(n-1)
         y = y + fib(n-2);
     return y;
}
```

Calculate pseudo-random integers with a linear congruential generator (LCG)

```
int lcong(
      const unsigned int a,
      const unsigned int b,
      const int n,
      const unsigned int s)
{
      unsigned int y = s;


      unsigned int sum = 0;
      for (int i = n ; i > 0; i--){
         y = y*a + b // calculate the new pseudo-random number
         sum = sum + y // add it to the total
      }
      return sum;
 }
```

Traverse a linked list to find an item

```
typedef struct item{
     int value;
     struct item *next;
} item_t;

item_t* find(const int x, item_t* head){
     while (head->value != x){
           head = head->next;
           if (head == NULL) break;
     return head;
}
```

Software requirements specification:

Functional requirements:

- Benchmark tests:
    - Fibonacci test and recursion
        - Stack
        - Stack pointer
    - LCG
        - Multiplication
    - Traverse linked list and find item
        - Indirect addressing
- Correctness
    - Use benchmark algorithms to check using trial data
    - Compare with hand-calculated results
- Speed - minimise geometric mean time (T1T2T3)^(⅓)
    - Found by counting number of CPU cycles required for each benchmark and how this changed with size of problem (e.g. size of list)
    - Find max clock speed of design and minimum execution time
- Power consumed - minimise number of logic gates
    - Number of logic gates and clock speed

Non-functional requirements:

- Greatest number of applications for least number of transistors
- 16-bit instruction word
- At least 2k words of instructions and 2k words of data
- Built and simulated using Quartus

5 Outline of technical problem

Technical Problem: Build a general CPU that can complete the given benchmarks.

The CPU was not a task to create a hyper specialised instruction set that only implements the benchmarks in the most efficient way and does nothing else. It also required the CPU to be Turing complete, which involve implementing indirect or register addressing of memory, computed jump (jump with register value), and loading current value of PC into a register or memory. Each benchmark also required specific instructions and hardware to be implemented, including subroutines, multiplication, and the stack.

6 Design Process

6.1 Overview of design

Memory

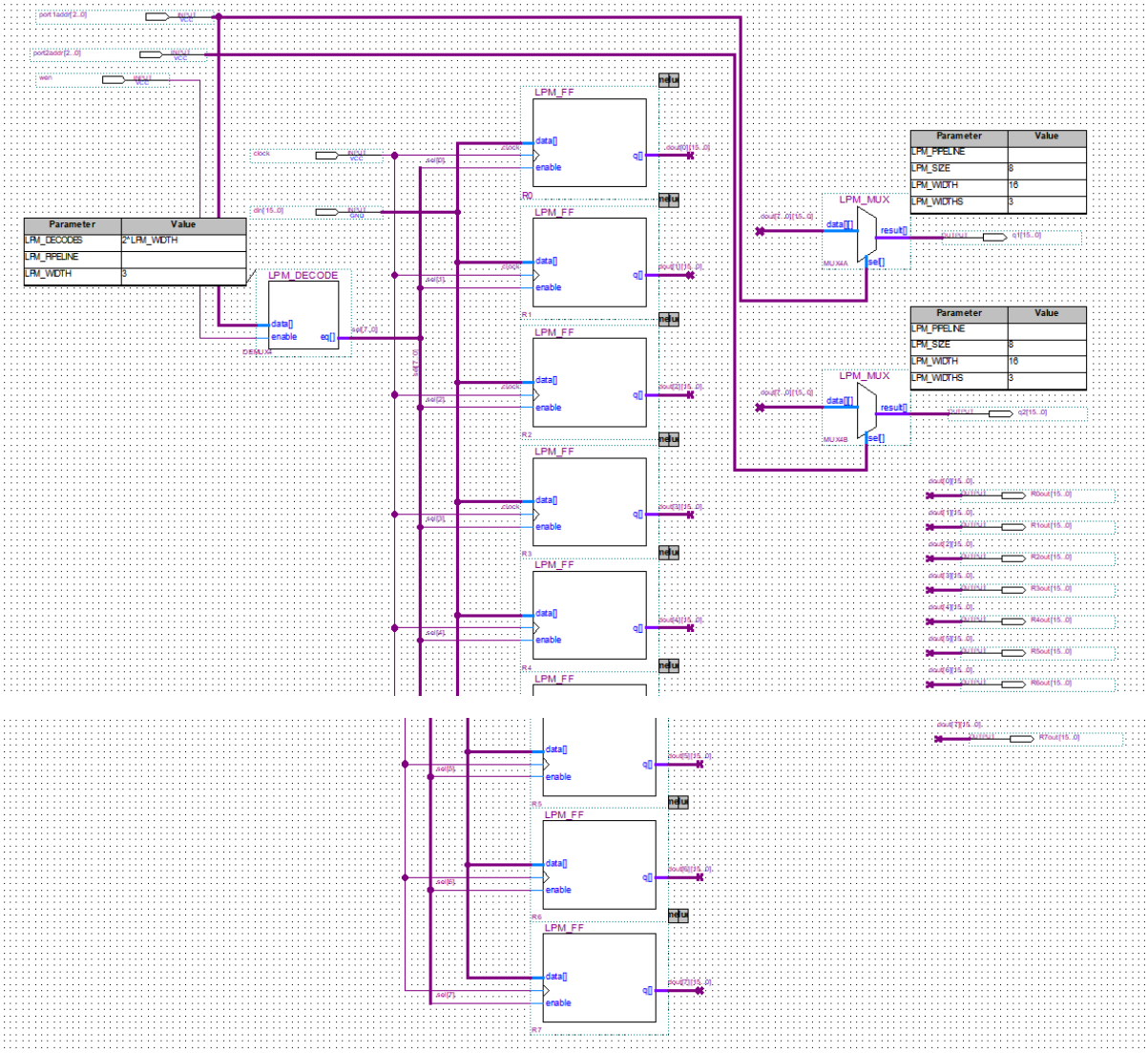
The memory blocks took inspiration from the Harvard architecture philosophy1. This involved separated the memory into an instruction ROM, where the main program was stored, and a data RAM, where all the data was stored. The different memory types could then be addressed differently. This was useful for implementing indirect addressing and a pointer to the stack (a section in the data RAM), as the ALU can output the correct data address for each cycle depending on the instruction. The instructions and data could also be accessed simultaneously, which was very useful for pipelining, as the next instruction can be fetched at the same time as reading or writing from the data memory.

The instruction memory could be either a ROM or RAM as the CPU never needs to write to this memory block. The two types of memory both simulate pretty much identically in Quartus. The ROM was used as it looked cleaner without all the extra write outputs for RAM (see Appendix 9 for more detail on decision).

The size of both memory blocks were 4096 16-bit words. As the opcode was 4 bits, one instruction (LDI) could use the rest of the bits to load a 12-bit constant that could be used as an address for all 4096 locations in the memory blocks. This was useful for testing.

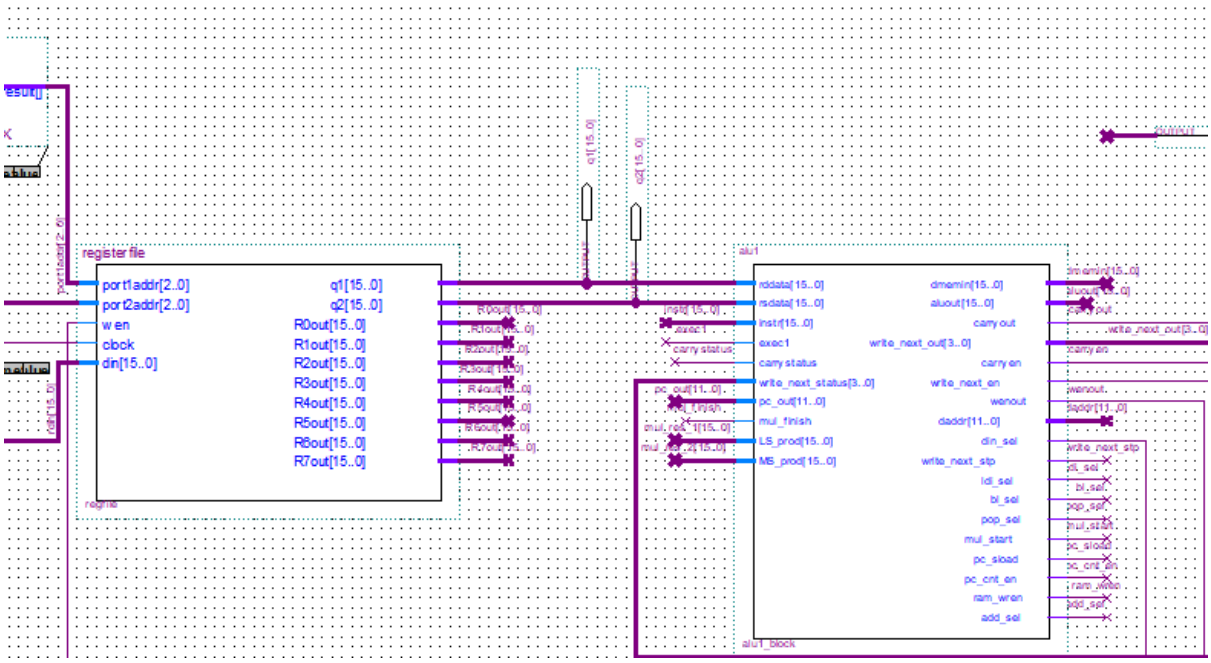
[1] Scott Thornton,"What's the difference between Von-Neumann and Harvard architectures" March 8th 2018 [online]MICROCONTROLLERTIPS available at: <https://www.microcontrollertips.com/difference-between-von-neumann-and-harvard-architectures/>

Register file



The register file included 8 registers. This required a 3-bit port addresses to select which register to read from or write to. Using 16 registers would have required 4 bits, which would take much more space in the instruction words, and many registers would not be used for the benchmarks given. This register file took inspiration from the ARMish CPU register file built in the second term, although register files are common for most CPUs.

ALU with register file (file called ALU1)



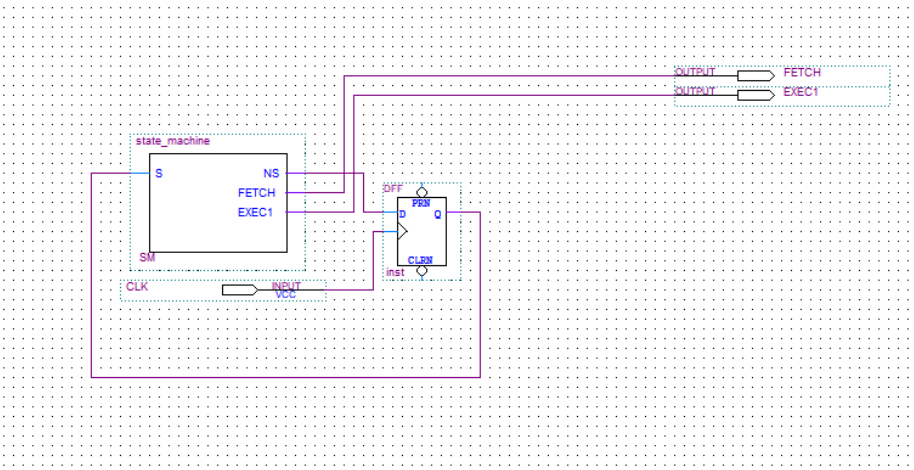
Like ARMish, the ALU was able to read from two registers in the register file, addressed using bits in the instruction word. The ALU implemented the decoder for all the signals and could do calculations on the two registers that were read. It handled all the logic and was implemented in Verilog (full ALU in Appendix 1).

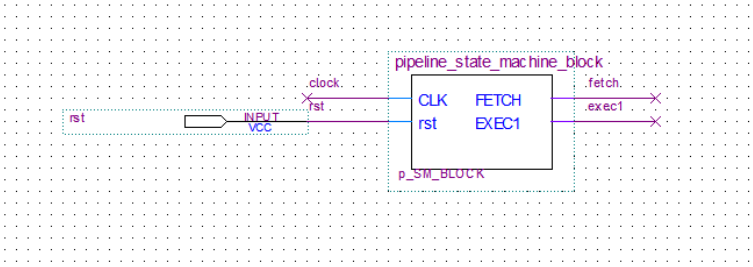
State machine (unpipelined)

The synchronous state machine controlled what state the CPU was in.

The unpipelined state machine just switched between the execute (EXEC1) cycle and the FETCH cycle as a new instruction is fetched every second cycle. This made it easier to pipeline the CPU as all logic was already carried out during EXEC1. The states were numbered as above, so the next state was just the inverse of the current state.

```
module state_machine
(
    input S,
    output NS,
    output FETCH,
    output EXEC1
);

assign NS = ~S;

assign FETCH = ~S;
assign EXEC1 = S;

endmodule
```





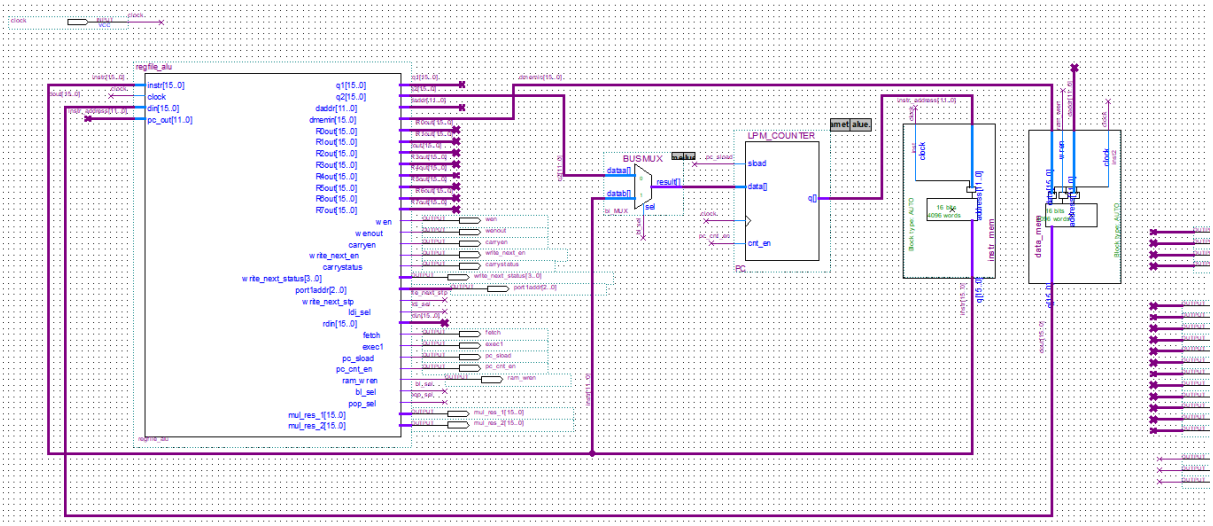
Decoder:

Originally, the decoder for control signals was separate from the ALU, although most logic signals were in the ALU, so many output and inputs needed to be created to leave different BSF files, which increased the time it took to test. This was simplified by moving the state machine into the regfile_ALU BDF and decoder logic was put into the ALU.

Original decoder:

```
module decode
(
    input [3:0] op,
    input FETCH,
    input EXEC1,
    output pc_sload,
    output pc_cnt_en,
    output ram_wren,
    output pc_in_sel,
    output ldi_sel,
    output din_sel
);

wire ldi, ldr, str, mov, jmp, add, sub, mul, xsr, xsl, bl, stp;
assign ldi = !op [ 3 ] & !op [ 2 ] & !op [ 1 ] & !op [ 0 ];
assign ldr = !op [ 3 ] & !op [ 2 ] & !op [ 1 ] & op [ 0 ];
assign str = !op [ 3 ] & !op [ 2 ] & op [ 1 ] & !op [ 0 ];
assign mov = !op [ 3 ] & !op [ 2 ] & op [ 1 ] & op [ 0 ];
assign jmp = !op [ 3 ] & op [ 2 ] & !op [ 1 ] & !op [ 0 ];
assign add = !op [ 3 ] & op [ 2 ] & !op [ 1 ] & op [ 0 ];
assign sub = !op [ 3 ] & op [ 2 ] & op [ 1 ] & !op [ 0 ];
assign mul = !op [ 3 ] & op [ 2 ] & op [ 1 ] & op [ 0 ];
assign bl = op [ 3 ] & !op [ 2 ] & !op [ 1 ] & !op [ 0 ];
assign ldmfd = op [ 3 ] & !op [ 2 ] & !op [ 1 ] & op [ 0 ];
assign stmfd = op [ 3 ] & !op [ 2 ] & op [ 1 ] & !op [ 0 ];
assign stp = op [ 3 ] & !op [ 2 ] & op [ 1 ] & op [ 0 ];


// Need to think of all decode signals and then implement logic
assign pc_sload = exec1 & jmp;
assign pc_cnt_en = exec1 & ~(jmp | stp);
assign ram_wren = exec1 & (str | stmfd);
assign ldi_sel = ~ldi;


endmodule
```

CPU block



This block made the connections between the regfile_alu block, the program counter and the memory. The program counter (PC) addressed the instruction memory, while the address for the data memory came from the ALU. The MUX at the data input of the PC switched between loading the PC with the value of the second register that was read for jump instructions and the least significant 12 bits of the instruction for branch instructions (discussed in the next section).

6.2 Instruction Set Architecture (ISA) choice explanation

The instructions were chosen to meet the requirements of the benchmarks and to create a general CPU. Given the word length of 16 bits, there was a limitation to how many bit fields could fit into the instruction. The register file contained 8 registers, which required a register address of 3 bits to select one of them. Therefore, two-operand instructions were used as it only took 6 bits of the instruction to select the two registers to read/write to. Three-operand instructions would be too large as they would require 9 bits and only 7 bits would be left for the opcode and other bit fields.

To decide upon the instructions required, the benchmark codes were first converted into a first draft of assembly code, so that the required instructions could be found.

ISA:

LDI	Load a 12-bit constant into R0
LDR	Load value from memory into a register
STR	Store value from register into memory
MOV	Move shifted value of one register to another
JMP	Jump to an address depending on a condition
ADD	Add two register values
SUB	Subtract two register values
MUL	16-bit multiplication of two register values stored in separate output registers
BL	Branch and link
LDMFD	POP - Load value from stack and increase stack pointer
STMFD	PUSH - Store value into stack and decrease stack pointer
STP	Stops the program

LDI: It was useful to load a 12-bit address into a register to be able to address all 4096 locations in memory. Due to the 16-bit word limit, LDI only worked for one instruction

LDR, STR: These instructions took inspiration from ARM, and were both used to implement register addressing. This is needed for the “find in list” benchmark.

MOV: Just like in ARM, the MOV instruction also carried out different shifts on the register value. While this was not specifically required for the benchmarks to work, it is a very useful function for general CPUs to have.

JMP: Jump was required for all benchmarks to implement loops in assembly code. The benchmarks also required conditional jumps, which were implemented using a condition field in the instruction that determines what condition needs to be met for the jump to work.

ADD, SUB: Based off ARMish.

MUL: Multiplication was required for the pseudo random number generator. This was implemented with a separate multiplier block, which both allowed the team to work in parallel, and for the ALU to carry out instructions at the same time as the multiplication is occurring.

BL: Branch and link[2] implemented subroutines, and were based off ARM. This involved having a link register (register 6) that stored the next address and then jumped to the address of the subroutine in the instruction word. When this subroutine (like a function in a separate part of the memory) was finished, it jumped to the value in the link register to return to where it left off in the main program. This instruction was useful when implementing the recursive Fibonacci benchmark test as it needed to call the Fibonacci function multiple times and needed to return to where it left off.

LDMFD/STMFD (POP/PUSH): These instructions implement the stack that is required for the Fibonacci benchmark. The stack was a section of the data memory addressed by a stack pointer. The stack pointer (register 7) decreased for POPs and increased for PUSHes. This took inspiration from the AVR ISA[3]. Each of these instructions needed to be called three times to pop/push the value of the link register, and the two local variables (M for multiple). The FD stands for full descending as the stack starts at the end of the data memory. This prevented the need to initialise the stack pointer at a particular address and run the risk of it exceeding 0xFFFF. Instead, the stack starts at 0xFFFF and descends when adding new items to the stack.

[2] Clarke, T. Class Lecture, Topic "LEC9.pdf", Department Of Electrical & Electronic Engineering, Imperial College London, UK,2020. Available at: <https://intranet.ee.ic.ac.uk/t.clarke/arch/html16/lect16/lec9.pdf>

[3] http://ww1.microchip.com/downloads/en/DeviceDoc/doc2503.pdf

6.3 ISA format

The decided ISA format took some inspiration from the ARM Thumb ISA4, due to the 16-bit word length. It helped make decisions on how many bits are needed for each field and how to reduce the instruction set.

ISA Table:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	0	0	12Bit constant												LDI
1	0	0	0	1	S	OFFSET					Rd			Rs			LDR
2	0	0	1	0	S	OFFSET					Rd			Rs			STR
3	0	0	1	1	CIN		cwen		SHIFT		Rd			Rs			MOV
4	0	1	0	0	Cond	Comparator					Rd			Rs			JMP
5	0	1	0	1	CIN		cwen				Rd			Rs			ADD
6	0	1	1	0	CIN		cwen				Rd			Rs			SUB
7	0	1	1	1							Rd			Rs			MUL
8	1	0	0	0	Address of the subroutine												BL
9	1	0	0	1							Rd			1	1	1	LDMFD
10	1	0	1	0							1	1	1	Rd			STMFD
11	1	0	1	1													STP

Note: Rd= destination register, Rs = source register

Descriptions:

0	0	0	0	12Bit constant	LDI

LDI would load a constant to Register 0. It would read the operand (12 bits) to store the integer to the register. Being able to load a 12 bits constant would allow the CPU to load any address number to the register, and thus it could jump to any address in the memory. The opcode of LDI is 0000, so this helped address directly to R0 by letting port address in the register file read from the opcode.

[4] ARM QRC 0006E, Thumb 16-bit Instruction Set Qucik Reference Card [online] available at: <http://infocenter.arm.com/help/topic/com.arm.doc.qrc0006e/QRC0006_UAL16.pdf>

0	0	0	1	S	OFFSET	Rd	Rs	LDR

LDR would load a value stored in the memory to one of the registers. Rs bits [2:0] was the address of the register which stored the address of the value to be loaded to another register. Rd bits [5:3] was the address of the register where the value should be moved to. OFFSET bits [10:6] would provide a positive or negative offset to the value in the Rs depending on the S bit. Sign bit (S) [11] would define the direction of the offset: 0 would be positive offset, 1 would be negative offset.

In order to do that, the value stored in the memory first need to be loaded into R0 using LDI.

The offset function helped the CPU implement register addressing for the "Finding in linked list" benchmark, since it could load the next location of the value where the address of the next item is stored.

The problem is that the LDR would require two cycles, one for the retrieving the data from RAM, one for writing back the value to the register. This was solved using the same state machine by allowing instructions to be completed in parallel (See Hardware descriptions in next section).

0	0	1	0	S	OFFSET	Rd	Rs	STR

STR was implemented in the same way as LDR except instead of loading a register with a memory address, STR stores the register value into the memory address given by mem[Rs ± Offset]. This was not required for the benchmark but helped generalise the CPU.

0	0	1	1	CIN	cwen	SHIFT	Rd	Rs	MOV

MOV would move the value from Rs to Rd while also doing optionally shifting the value. Rs bits [2:0] was the register which stored the value to be moved. Rd bits[5:3] was the register where the value should be moved to.

Shift type:

00: no shift

01: shift left

10: shift right

11: move multiply registers

CIN field:

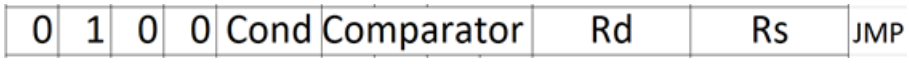
00: cin=0

01: cin=1

10: cin = carrystatus (previous carry)

11: cin = CMSB (carry most significant bit of Rs)

The Move multiply registers option was added later with multiplication as the product is stored in two separate registers from the register file, so these instructions were necessary to move them to the register file. Only one MOV instruction was required after multiplying for the LCG as only the least significant 16 bits are required. Although, general multiplication with the full 32-bit product is possible.



JMP would change the address in the program counter under different condition. Rs bits [2:0] was the register which stored the address the program counter should jump to. Rd bits [5:3] was the register which stored the value to be compared with the comparator. Comparator bits [9:6] was the value to be compared with the value in Rd.

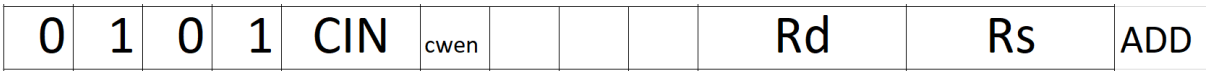
COND field [11:10]

00: JMP (always)

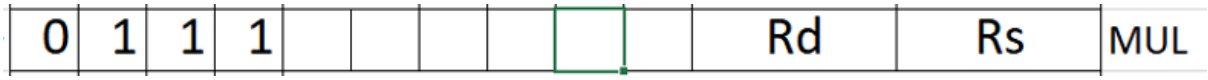
01: JEQ (jump if equal) Rd==comparator

10: JMI (jump if less than) Rd < comparator

11: JMB (jump if bigger than) Rd > comparator



ADD/SUB would add/subtract the value in Rd and Rs and store the result back to Rd. Rs bits [2:0] was the register which stored one of the number for addition. Rd bits [5:3] was the register which stored one of the number for addition and the register the result would be stored in. Bits [8:6] are reserved for future features. CWEN bit [9] would enable writing the result to the carry register. CIN is added to the result. Note that normal subtraction required CIN=1 as the conversion to 2's complement of Rs required an inversion and addition of 1.



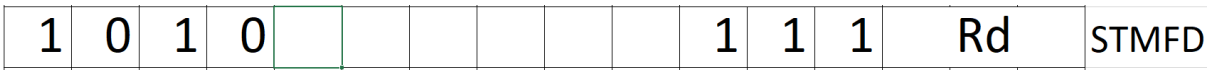
MUL carried out 16-bit multiplication between Rd and Rs and stored the product in 2 output registers after the multiplier block.

LDMFD/POP would load the value stored in the stack area back to the register. Bits [2:0] was the location of the stack register (R7). Rd bits [5:3] was the register where the value should be restored to. Bits [11:6] were not needed.

The process of this instruction is

Rd = Mem[Stack Pointer]

Stack Pointer = Stack Pointer + 1



STMFD would store the value in the register to the stack area. Rd bits [2:0] was the register stored the value to be preserved. Bits [5:3] was the location of the stack register (R7). Bits [11:6] were not used. The location of 111 was flipped as it made the logic easier.

The process of this instruction is:

Stack Pointer = Stack Pointer - 1

Mem[Stack Pointer] = Rd

The reason why stack pointer would decrement by one first is that the stack register would initialise to be 0, and minus one would let it be 0xFFFF which pointed to the end of the memory.

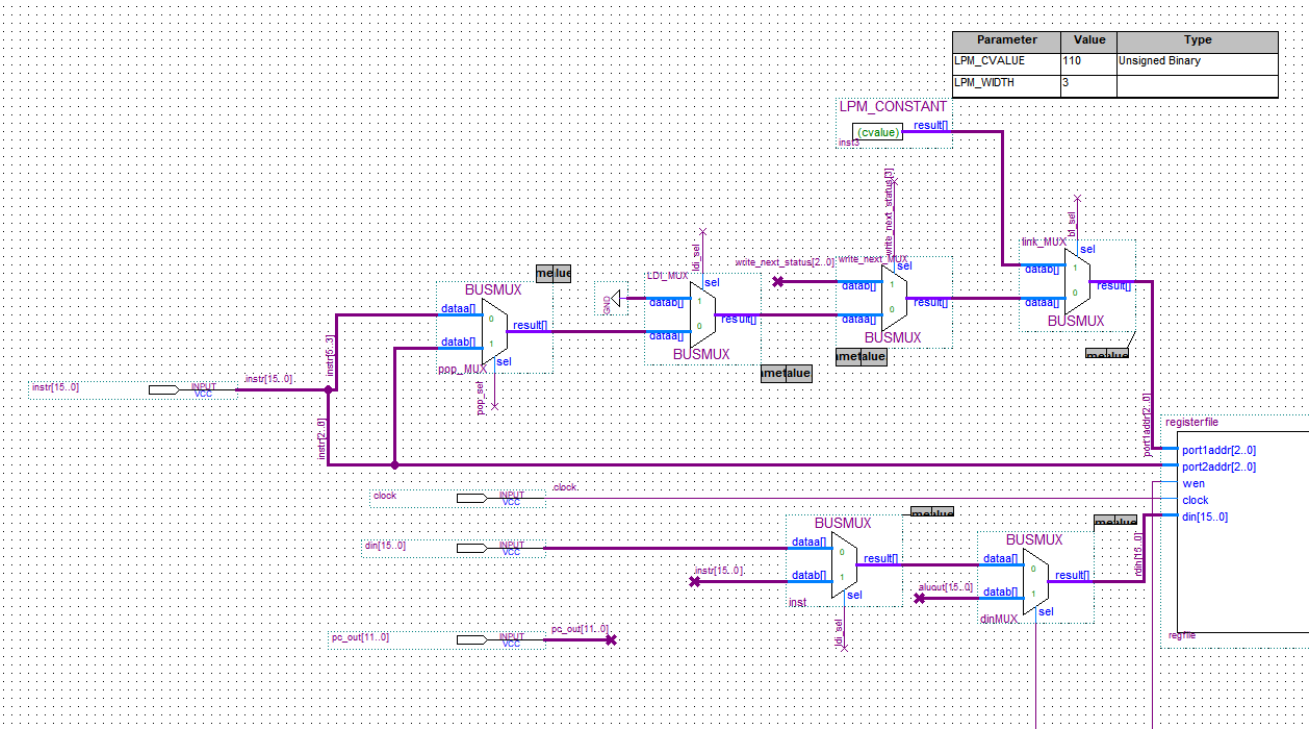
1	0	1	1												STP

STP function would stop the whole program (prevent the program counter from counting).

6.4 Instruction Hardware Implementation

LDI, ADD, SUB, and STP were implemented like ARMish

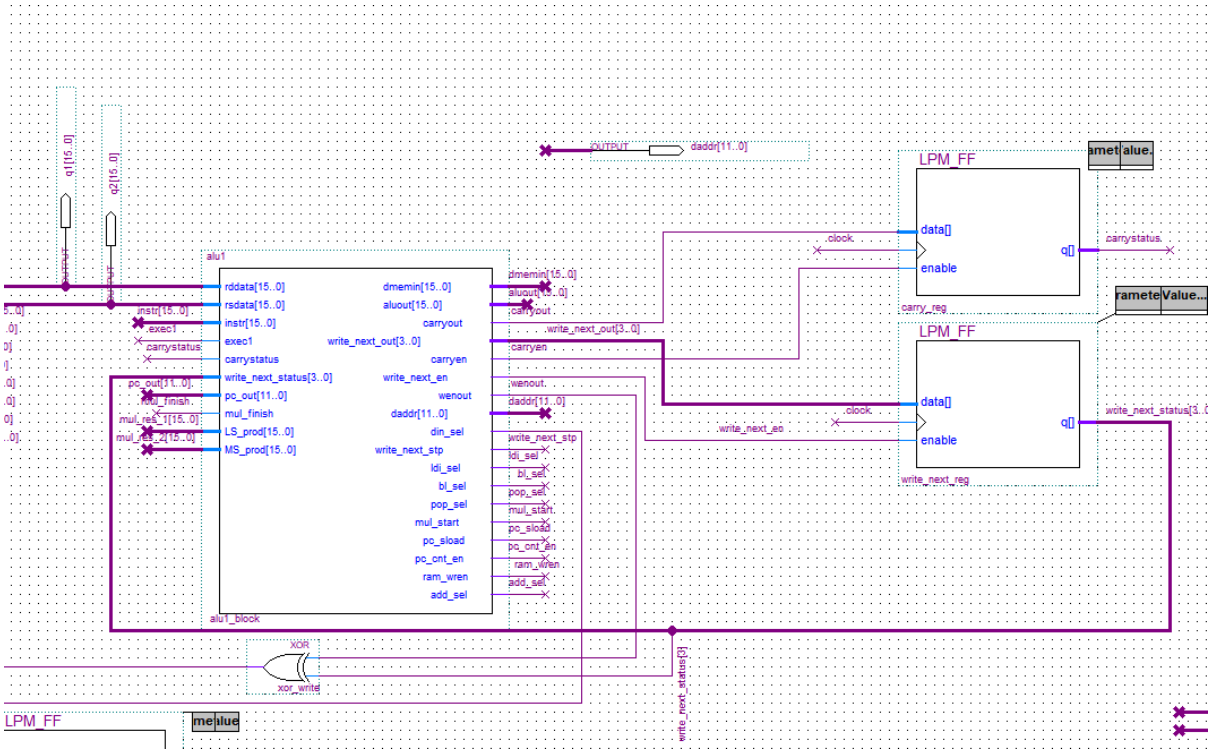
Input to register file



The series of MUXes selected the input of port1addr[2:0] (to address the register to write to) depending on the instruction. The possible inputs included instr[5:3] to address Rd, instr[2:0] for selecting register 7 for the pop instruction, 3'b000 for LDI, 3'b110 (register 6) for writing into the link register in BL, and write_next_status[2:0], which is used for LDR (described

below). The MUXes at the input of din (data in of register file) also switched between either data from the RAM (din[15:0]), the instruction word (instr[15:0]), or the output of the ALU (aluout[15:0]). See appendix 1 for ALU logic for control signals.

Output of ALU



The carry flip-flop bits carryout, carryen, and carrystatus were implemented using the ARMish design. The other register is used for LDR instructions as explained below.

LDR hardware implementation

LDR required two cycles: one for reading a value from the RAM, and one for writing data out of the RAM into the register file. This meant that data in (din) of the register file is only updated one cycle after the LDR instruction, so port1addr needs to be correct one cycle afterwards, as well as write enable (wen). A register called write_next was added at the ouput of the ALU that stores “write_next_flag” (telling the next instruction that the data is now at dout during the next cycle and can enable writing) and the address of Rd given by bits [5:3] of the LDR instruction. This register was enabled every execute cycle.

```
// Status FF bits:

// write_next tells next instruction that the data that is now at dout can be written into the Rs of the previous instruction (for load instructions)
// if write_next_flag is already 1, set to 0 (if it is not another ldr), otherwise set to 1 if it is an ldr or ldmfd instruction
assign write_next_flag = (write_next_status[3] & (~ldr | ldmfd)) ? 0 : (ldr | ldmfd);

// write_next_en enables writing to write_next register - needs to update for every instruction during exec1 so that it returns to 0
assign write_next_en = exec1;

// Output to write_next register
assign write_next_out = {write_next_flag, instr[5:3]};

// write_next_stp stop PC from counting up so instruction can finish before next instruction
assign write_next_stp = write_next_status[3] & ~(ldr | str | jmp | ldmfd) | write_next_status[3] & ldmfd;


// carryen enables writing to carry register - writes when cwen is enabled for add, sub, and mov
assign carryen = exec1 & cwen & (add | sub | mov);

// carryout equal to alucout if not a shift - note the special case of rsdata[0] for LSR or ASR (MOV instruction)
assign carryout = (mov & ((~field[1] & field[0]) | (field[1] & ~field[0]))) ? rsdata[0] : alucout;
```

This implementation allowed other instructions to be carried out in parallel with this second cycle for LDR, given that the following instruction does not write to the registers or read from the register being written to by LDR (as it has not updated yet). To prevent the CPU from bugging if the next instruction was a write instruction, the write_next_stp bit stops PC from counting during this second LDR instruction to allow it to finish before the next instruction executes. This acted like a delay to the system and allows the assembly to have any instruction after LDR as the ALU decides if the PC waits for LDR to finish or if the next instruction can be done in parallel. Note that LDR instructions could be executed after each other and be done in parallel as the first cycle of LDR just involved the data RAM. An XOR gate between the write enable out (wenout) of the ALU and the write_next_flag from the write_next register determined if the register is written or not (as these two bits should not be on at the same time).

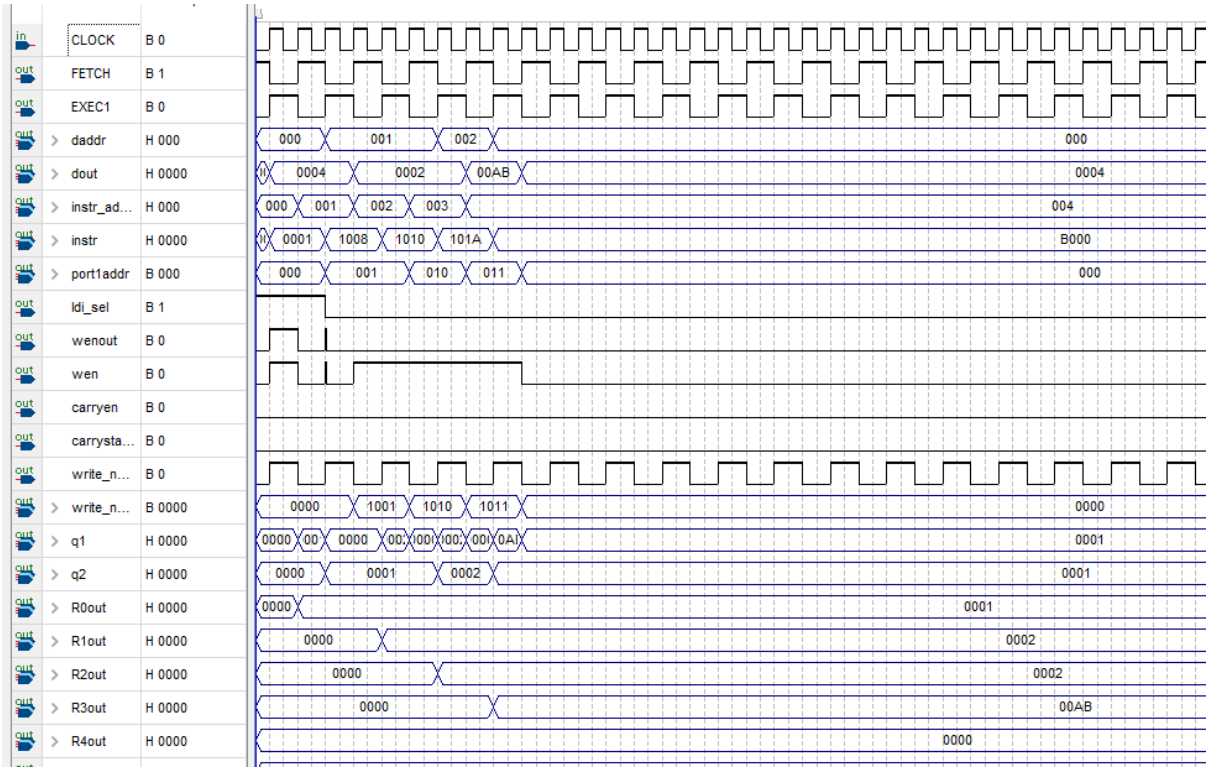
LDR instructions after each other

Data memory:
0x0004
0x0002
0x00AB

Instruction mem:

LDI 0x1	0001
LDR 0 0 R1 R0	1008
LDR 0 0 R2 R0	1010
LDR 0 0 R3 R2	101A
STP	B000

Simulation:



In the simulation multiple LDR instructions could be executed consecutively after each other as the output of a register updated every EXEC1 clock edge.

Indirect addressing (LDR and STR)

```
// ALU output Calculations
assign alucout = alusum[16]; // carry bit
assign daddr = ldmfd ? (alusum[11:0]-1) : alusum[11:0]; // offset address used for addressing data memory or stores R7 value for stack manipulation
assign aluout = alusum[15:0]; // 16 bit sum

assign dmemin = stmfd ? rsdata : rddata; // input to data memory is q2 for stmfd, q1 for str
```

```
// determine alusum - need to change opcodes
always @(*) begin
      case (op)
      4'b0001, 4'b0010 : alusum = sign ? (rsdata - {11'b0,offset}) : (rsdata + {11'b0,offset}); // LDR and STR: calculate daddr
```

Indirect addressing was achieved by calculating the offset address into alusum and storing this as the data address (daddr) as seen above. The offset address was calculated using the parameters specified in the instruction word. This was used in both LDR and STR to specify the address for the RAM.

MOV

The move instruction was implemented similarly to ARMish together with the shift functionality inside the ALU using alusum.

```
4'b0011 : begin
            case (instr[7:6])
            2'b00 : alusum = {1'b0,rsdata} + cin; // MOV
            2'b01 : alusum = {rsdata,cin}; // LSL
            2'b10 : alusum = {rsdata[0], cin, rsdata[15:1]};// Right shift - cin determines LSR or ASR
            2'b11 : alusum = instr[0] ? ({1'b0,MS_prod}+{1'b0,rddata}+cin) : ({1'b0,LS_prod}+{1'b0,rddata}+cin);
            default : alusum = {1'b0,rsdata} + cin; // MOV
          endcase;
        end
```

The option of no shift also could add cin, so this was a quick way of moving a register and adding one simultaneously. When instr[7:6]=11, the mov instruction output either the least significant 16 bits (LS_prod) or the most significant 16 bits (MS_prod) of the multiplication result depending on whether Rs is 0 (moves LS_prod) or 1 (moves MS_prod). LCG only used the Rs=0 option, as only LS_prod is required. Alusum is directed back into the data in of the register file to be written into the register specified by port1addr.

JMP:

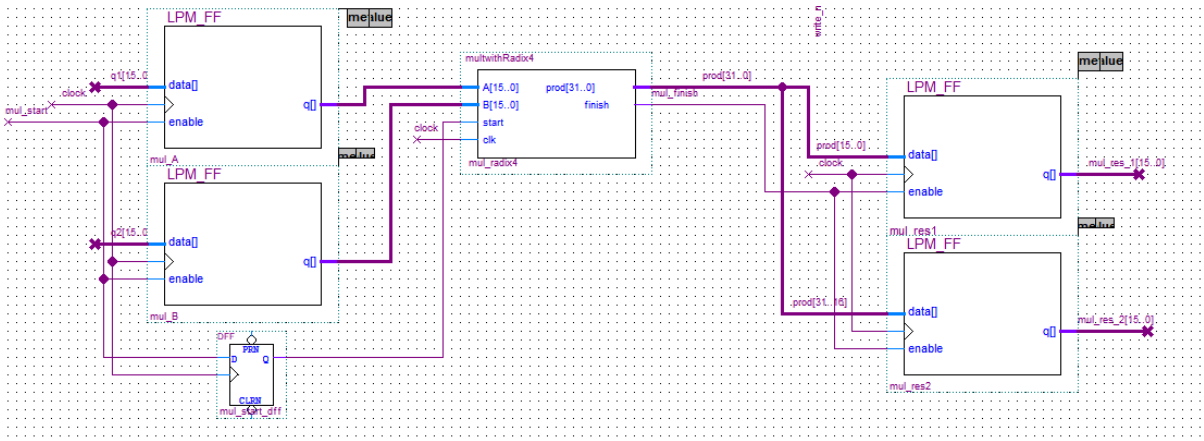
On the other hand, jump was implemented with a wire, jmp_cond, that determined if the PC should load the address of Rs depending on the conditions given in the opcode and by the comparisons made (eq and mi). Jmp_cond enables PC loading and stops it from counting.

```
// conditional operators - compares Rd and comparator (in cond) - used for if jump should occur
wire eq = (rddata == cond); // Rd == comparator
wire mi = (rddata < cond); // Rd < comparator

// change jmp cond to determine if jmp will occur depending on comparison and jump condition in instruction word
wire jmp_cond = jmp & ((~field[1] & ~field[0]) | (~field[1] & field[0] & eq) | (field[1] & ~field[0] & mi) | (field[1] & field[0] & ~eq & ~mi));

// PC and RAM control signals
assign pc_sload = exec1 & (jmp_cond | bl);
assign pc_cnt_en = exec1 & ~(jmp_cond | stp | bl | write_next_stp);
assign ram_wren = exec1 & (str | stmfd);
```

MUL:

Refer to multiplication method research for information about the Booth Radix 4 multiplier that was used.

Hardware in regfile_alu bdf:



The left-hand registers were used to store the values read from the two registers in the register file specified in the instruction word. This was necessary as these multiplicands needed to be constant throughout the entire 9 cycles that it takes this multiplier to carry out the multiplication. When the MUL instruction started, the mul_start control signal was asserted, getting stored in a DFF, so that in the next cycle, the multiplication can start in the multwithRadix4 block. When it was done, the mul_finish signal was activated, which enabled writing to the output registers, where the 32-bit product was stored. As discussed previously, the MOV instruction had an option to store this product into the register file. This could not be done with the MUL instruction as there was no space in the instruction word to include output registers.

```
2'b11 : alusum = instr[0] ? ({1'b0,MS_prod}+{1'b0,rddata}+cin) : ({1'b0,LS_prod}+{1'b0,rddata}+cin);
```

In the alusum calculation, add and multiply was incorporated into this design, so that the multiply product gets added onto what is already in the destination register. This allowed it to first move “b” into the register in the LCG benchmark and then move the product into the same register, where it automatically adds the “b”. Therefore, an additional ADD instruction was not needed.

BL: Branch and Link

The link register was stored with “PC+1” and the port1addr selects 6 the MUX described at the beginning of the section.

```
4'b1000 : alusum = {5'b00000,pc_out} + 1; // BL
```

When the MUX at the data in of PC was asserted, it selects the least significant 12 bits of the instruction to be loaded into the PC. This carried out the jump from the instruction word rather than from a register for JMP instructions.

LDMFD(POP)/STMFD(PUSH)

LDMFD (pop) writes the new value into the stack pointer (subtracted by 1) while using the old value as the data address. STMFD (push) writes the new value into the stack pointer (added by 1) while using the new value as the data address. This logic was reflected in the daddr wire seen below. The stack pointer, R7 was selected at port1addr using a MUX (either from Rs or Rd – refer to ISA table).

```
4'b1001 : alusum = {1'b0,rddata} + 1; // POP (LDMFD)
4'b1010 : alusum = {1'b0,rddata} - 1; // PUSH (STMFD)
```

```
assign daddr = ldmfd ? (alusum[11:0]-1) : alusum[11:0]; // offset addressed used for addressing data memory
```

All tests for instructions found in Appendix 13

6.5 Multiplication method research

Multipliers are important in CPU design, as they have long latency and consume relatively considerable power. A system's performance largely depends on multipliers because the multipliers are one of the slowest element within the system. In this section, a few ways of implementing the multipliers have been introduced and the strength and drawback of each implementation is discussed. In real life, the most important thing about multipliers is power, the power consumption of multipliers need to be controlled as low as possible, and proper cooling system need to be designed to remove the heat generate from the multiplier. However, in this project, the improvement in fundamental arithmetic functionality and speed of the multiplier can outweigh the increased power usage. Below is a table that shows multiplication in math perspective.

Multiplication table

$$
\begin{array}{rcccc}
 & A_3 & A_2 & A_1 & A_0 \\
\times & B_3 & B_2 & B_1 & B_0 \\
\hline
 & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 \\
+ & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 \\
+ & A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 \\
+ & A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 \\
\hline
\end{array}
$$

Multiplying A(N-bits) and B(M-bits) resulted in M*N bits

Multiplier Type I : shift adding multiplier(Combinational Multiplier)

A shift and add multiplier architecture is demonstrated in Figure 3. The inputs include a start signal, clock, and multiplicand. It used the shift and add method to output the result with a stop signal.

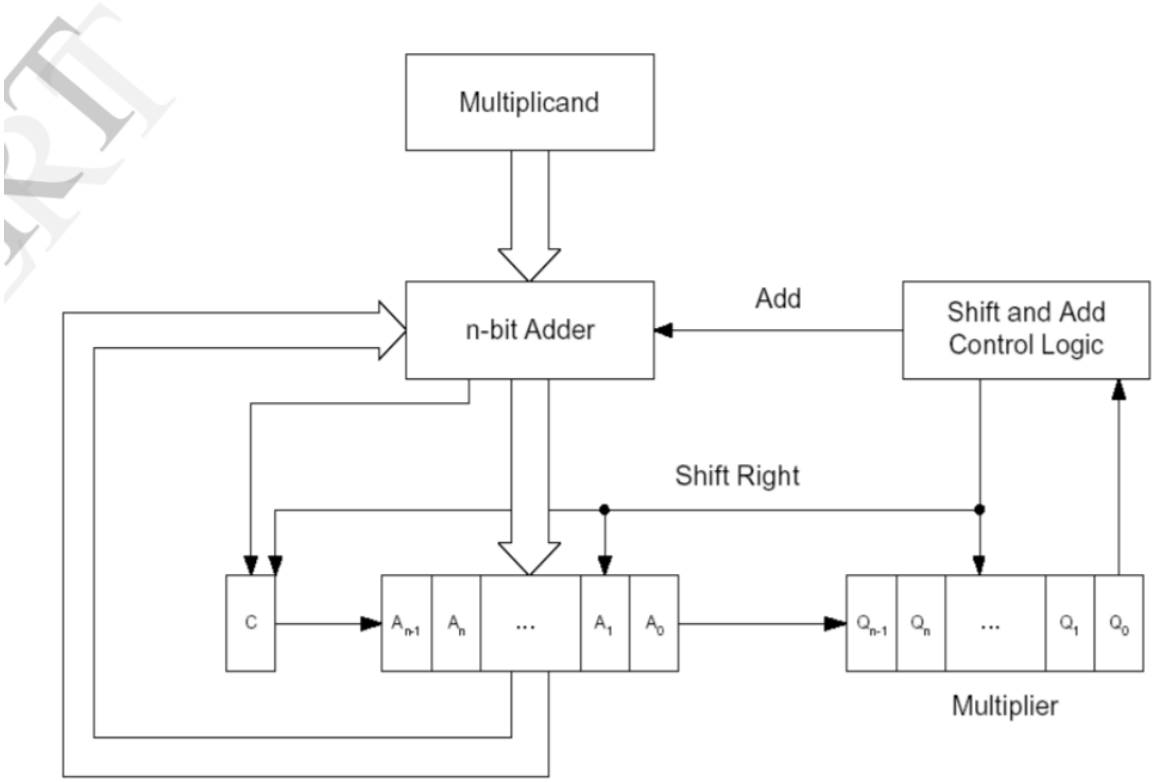
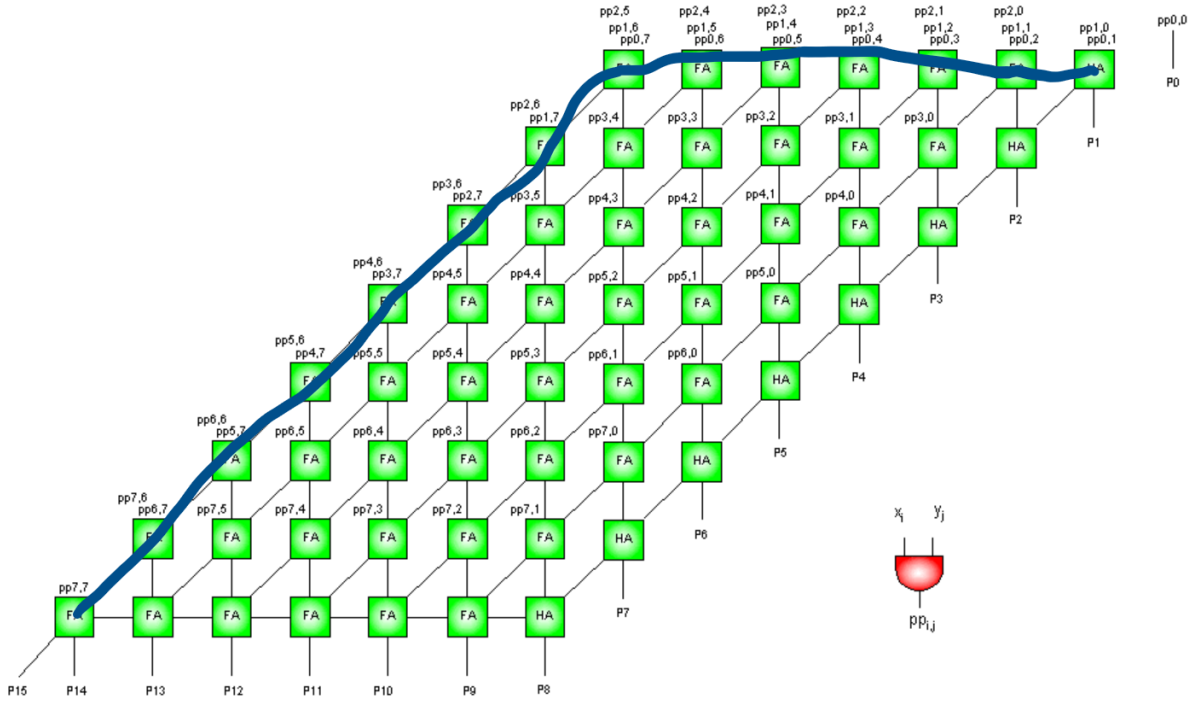


Figure 3[5]

These multipliers are simple and take up little area. Higher radix multipliers are quicker, although there is a higher power usage due to the larger register use and more complicated

[5] Deepak Bordiya, and Lalit Bandil, "*Comparative Analysis Of Multipliers*" International Journal of Engineering Research & Technology. Volume 2 Issue 9, September - 2013, pp. 1437– 1441. Accessed on: 8 June 2020. [Online]. Available at: <https://www.ijert.org/research/comparative-analysis-of-multipliers-serial-and-parallel-with-radix-based-on-booth-algoritham-IJERTV2IS90625.pdf>

logic. Implementing the example above using Verilog, a time analysis could be run to find out the propagation delay. If the multiplier with full-adder and logic gate was implemented, it would contain many adders and logic gate. The logic is the same with the multiplication table, each individual bit of the product came from multiple addition. The exact number of additions depends on the bit number.



Figuer 4 http://www.ellab.physics.upatras.gr/~bakalis/Eudoxus/CSAM.html

This image illustrates a hardware implementation for an 8 * 8-bit combinational multiplier, if N is the bits of the multiplicand, the delay should be 2N times the delay of the full adder.

The implementation could be done in Verilog, although this would be the function already implemented in Figure 5.

```
wire [9:0] a,b;
wire [19:0] result = a*b; // unsigned multiplication!
If you want Verilog to treat your operands as signed
two's complement numbers, add the keyword signed to
your wire or reg declaration:
wire signed [9:0] a,b;
 wire signed [19:0] result = a*b; // signed multiplication!
```

Figure 5 [6]

Using the FPGA shown in Figure 6, the propagation delay would be around 10ns.

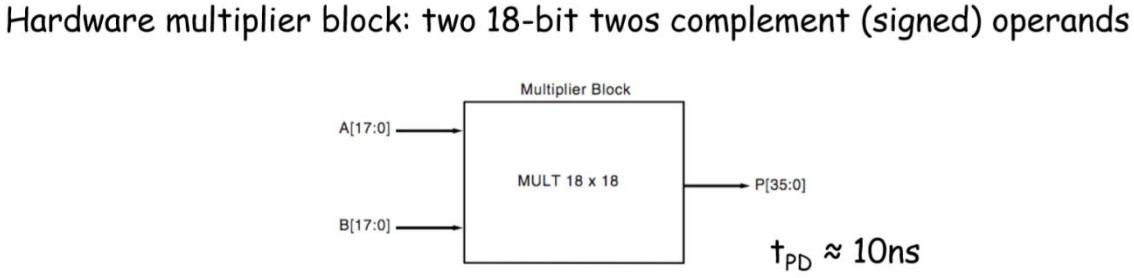


Figure 6 [6]

[6] Gim P. Hom, Joe Steinmyer, Class Lecture, Topic: "*Arithmetic Circuits & Multipliers*" 6.111 Introductory Digital Systems Laboratory, Massachusetts Institute of Technology, MA, Fall, 2017.Avaiable at: <http://web.mit.edu/6.111/www/f2017/handouts/L08.pdf>

Multiplier Type II Sequential Multiplier

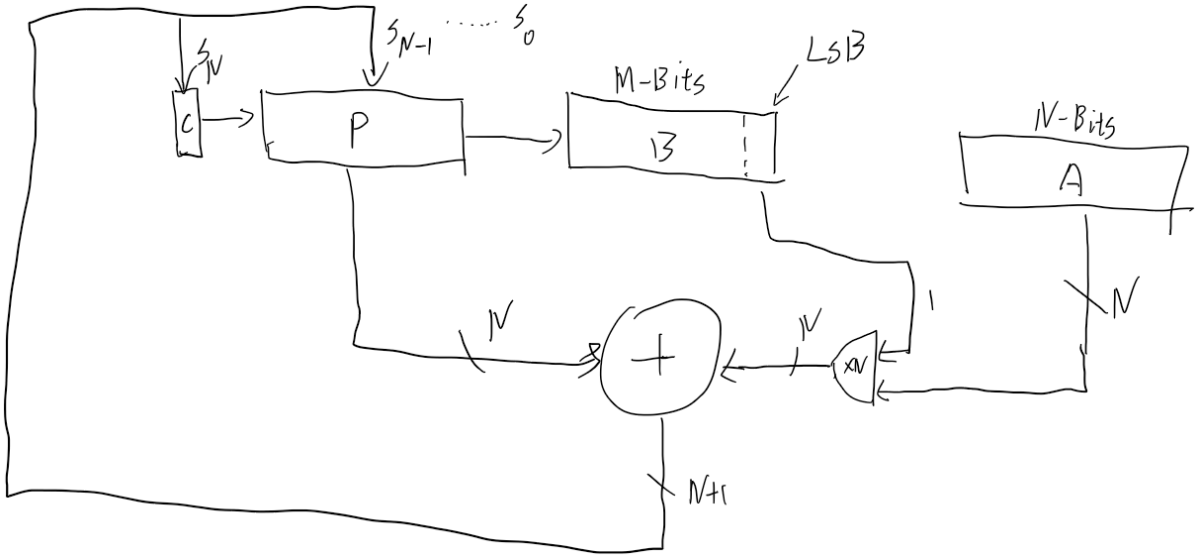


Figure 7

After the combinational multiplier, the sequential multiplier (Figure 7) was also commonly used in early hardware design. It only contains one full adder, four registers in total. A stored a M-bits multiplier, B store a N-bits multiplicand, P is N bit register and C is a single bit register. P and C had initial value of 0. To illustrate the multiple process, an example of 11*11 has been done in Figure 8.

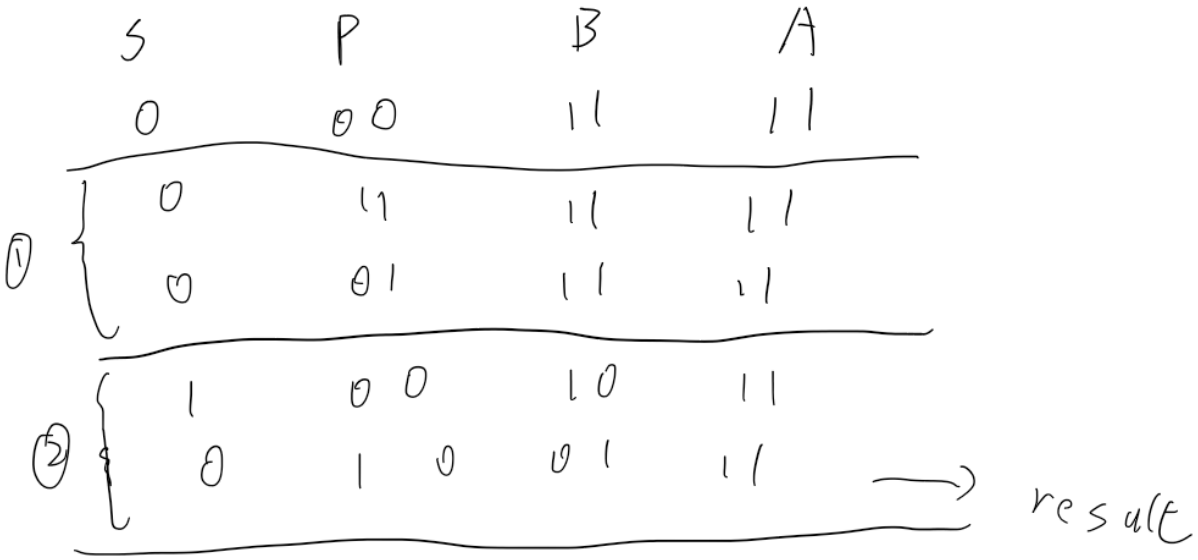


Figure 8

It was doing implicit shifting and adding operation, where P/B are shifting registers, they always shifted right by one each cycle. The mathematical explanation is shown in Figure 9. The shifting operation is determined if shifting and adding both needed to be done or just shifting. The addition step is added at the MSB side, which means adding multiplicand*2^N, N is the number of bits of the multiplier. However, at the end of the multiplication, the value stored in P/B is shifted to right by N bits. The shifting cancels out the effect of adding to the MSB.

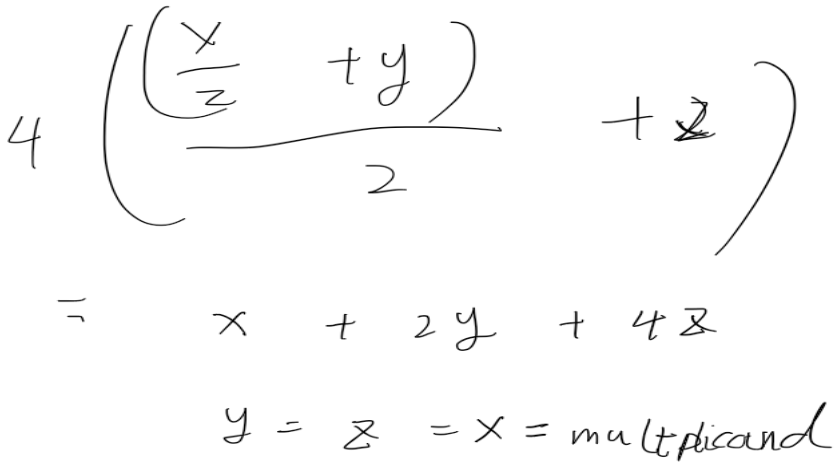


Figure 9

Multiplier Type III: Multiplier with Radix 4 and Radix 8.

The radix determines the addition choices in this case. In radix 2 multiplier, the only choice of addition was to add the multiplicand or not add the multiplicand. In radix 4, the choice for addition was not only 1 or 0 but 2*multiplicand and 3*multiplicand. In radix 8 the choices are even more. In radix 2 multiplier 16 additions need to be done for 16 bits multiplication, but with radix 4, only 8 additions need to be done and with Radix 8 it is 4 addition. Normally each addition takes one cycle, hence the number of cycles required could be reduced by increasing the Radix.

For Radix 4 multiplier, Figure 10 in Appendix, the shifting and adding method has also been used, the multiple of the multiplicand have been pre-computed before the additions. Instead

of shifting one bit to the right per cycle, 2 bits were shifted in Radix4 multiplier. The product of this 2 bits number and the multiplicand was added to with the product registers which is the left side to the register B, however it was shifted to the LSB at the end of the multiplication. So, the MSB partial product was not shifted because it is added at the end without any further shifting. Comparing to the normal Radix 2 multiplier, the Radix 4 take 9 additions, one of them was pre-computing the multiple of the multiplicand, it was more efficient than Radix2, the detailed tests regard to latency and power have been discussed in the optimisation section. In theory, higher radix multipliers have more power consumption and lower latency.

The waveform simulation has proved that the cycles required for multiplication have been reduced. Figure 11.

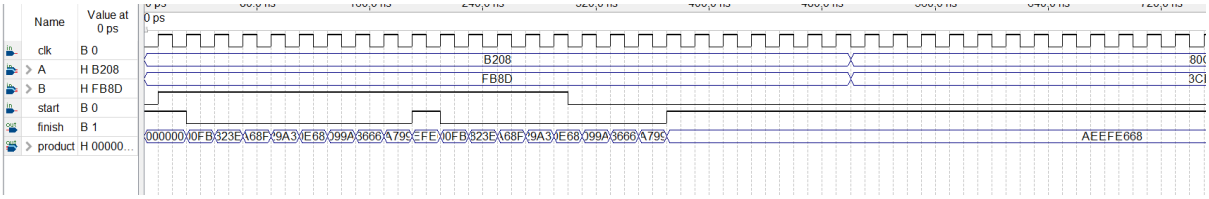


Figure 11

Radix 8 multiplier has the similar structure, Figure 11, but instead of shifting 2 bits, 4 bits were shifted. Therefore, the number of additions had been reduced further.

For the Radix 8 multiplier, more values needed to be pre-computed before the multiplication. However, the multiplication itself only took 4 additions. From the waveform simulation, Figure 12, the result would be available after 4 cycles, however the total number of additions of the Radix8 multiplier were 15 additions in total including 11 pre-computing additions. Clearly Radix 8 multipliers have larger propagation delay due to larger number of additions at the start compared to Radix 4 multipliers.

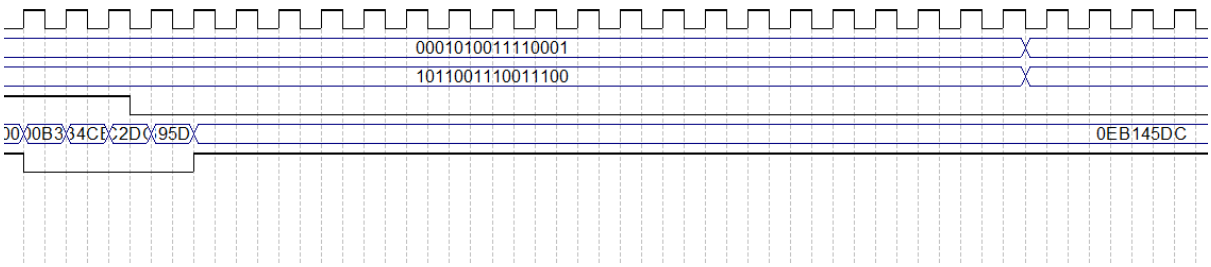


Figure 12

To conclude, higher radix multiplier meant less additions during the multiplication, but more pre-computed values were required. In this case, radix 8 multiplier need to do 6 more additions for one multiplication compared to radix 4 multiplier. For a 16bits or 32bits word, Radix 4 multiplier would be efficient enough. The radix of the multiplier should be chosen according to the bit length of operands. For the combinational multiplier, it used the most adders but could compute the result with less cycles, and the propagation delay problem could be solved by changing the intermediate adders to be carry save adders, so the carry would be saved instead of added. This would reduce the propagation delay of adding the carry. The final adding stage of the combinational multiplier would involve adding the carries. For the sequential multiplier, it could not be improved and had to take 16 cycles, but it only needed one full adder to do the multiplication.

7 Benchmark tests

7.1 Benchmark: Fibonacci numbers using recursion

Assembly:

R5 is the return register	R6 is the Link Register	R7 is the Stack register			
In Detail		Address	Opcode	Oprand	HEX value
R0 = 2	LDI 2	0	0000	000000000010	0002
R1 = n = 2	MOV R1 R0	1	0011	000000001000	3008
Call Fib(2)	BL Address of FIB	2	1000	004	8004
Stop	STP	3	1011		B000
R0 = 0	LDI 0	4	0000	000000000000	0000
R2 = 0 (y = 0)	MOV R2 R0	5	0011	000000010000	3010
	LDI ELSE's Address	6	0000	ELSE's Address	000C
if (R1>1) JUMP to ELSE	JMPB 1 R1 R0	7	0100	11 0001 001 000	4C48
R0 = 1	LDI 1	8	0000	000000000001	0001
R2 = y = 1	MOV 0 0 0 R2 R0	9	0011	000000010000	3010
Jump to END	LDI END's Address	10	0000	END's Address	0020
	JMP 0 0 R0	11	0100	000000000000	4000
ELSE	STMFD R6	12	1010	000000111110	A03E
Store all the current value	STMFD R1	13	1010	000000111001	A039
	STMFD R2	14	1010	000000111010	A03A
	LDI 1	15	0000	000000000001	0001
R1 = R1 - 1	SUB 0 0 0 R1 R0	16	0110	010000001000	6408
Call Fib(1)	BL Address of FIB	17	1000	FIB's Address	8004
Restore the values form the stack	LDMFD R2	18	1001	000000010111	9017
	LDMFD R1	19	1001	000000001111	900F
	LDMFD R6	20	1001	000000110111	9037
R2 = R5	MOV R2 R5	21	0011	000000010101	3015
	STMFD R6	22	1010	000000111110	A03E
	STMFD R1	23	1010	0 00000 111 001	A039
	STMFD R2	24	1010	0 00000 111 010	A03A
	LDI 2	25	0000	000000000010	0002
R1 = R1 - 2	SUB 0 0 0 R1 R0	26	0110	010000001000	6408
Call Fib(0)	BL Address of FIB	27	1000	FIB's Address	8004
	LDMFD R2	28	1001	0 00000 010 111	9017
	LDMFD R1	29	1001	0 00000 001 111	900F
	LDMFD R6	30	1001	000000110111	9037
R2 = R2 + R5	ADD 0 0 0 R2 R5	31	0101	000000010101	5015
END	MOV R5 R2	32	0011	000000101010	302A
PC = Link Register	JMP 0 0 R6	33	0100	000000000110	4006

Explanation:

Phase One: Calling

Loading the parameter n in to R1, and call the Fibonacci function.

Phase Two: Initializing variable y

Assigning local variable y to R2 by letting R2 = 0.

Phase Three: IF statement

Comparing whether n is bigger than one.

Phase Four: NOT Bigger

Assigning variable y to 1 by letting R2 = 1

Jump to the END phase

Phase Five: Bigger

Store all the values in this call to stack and call the Fibonacci (n-1)

Restore all the values and move the value in the return register(R5) to variable y (R2)

Store all the values in this call to stack again and call the Fibonacci (n-2)

Restore all the values and adding the values in the return register and variable y, and storing the result back to variable y.

Jump to END phase

Phase Six: END

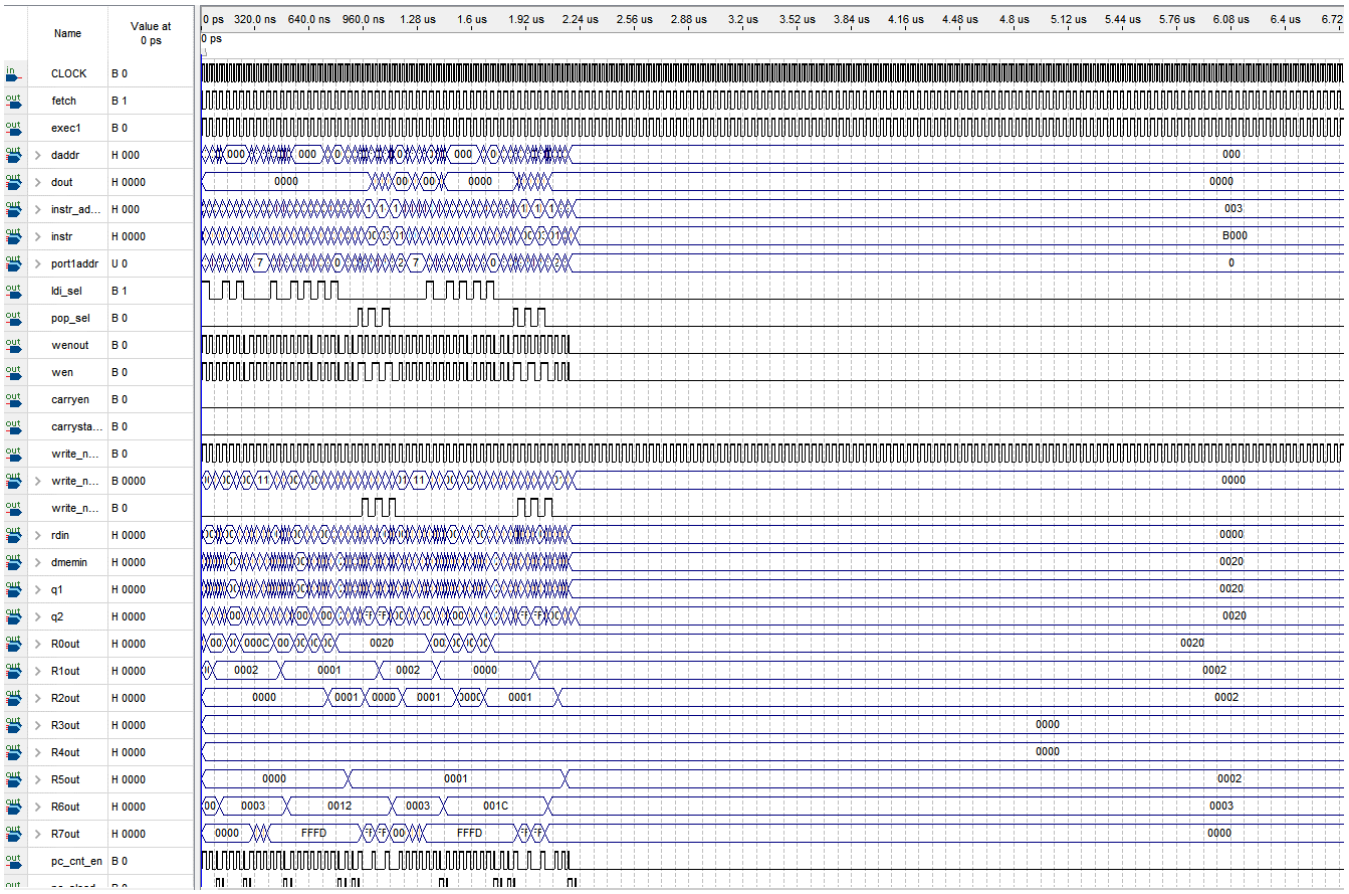
Move the value of variable y (R2) to the return register (R5)

Jump to the address stored in the link register

Testing:

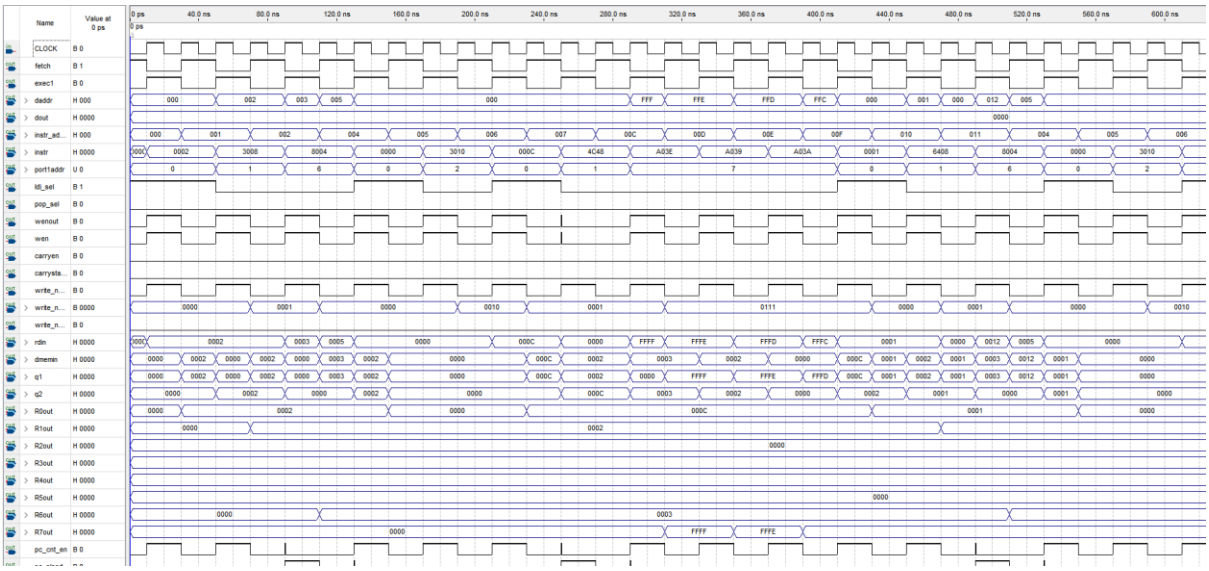
Fib(2)

107 cycles



The final value of R5 (the return register) was 2, which was correct for fib(2)=2.

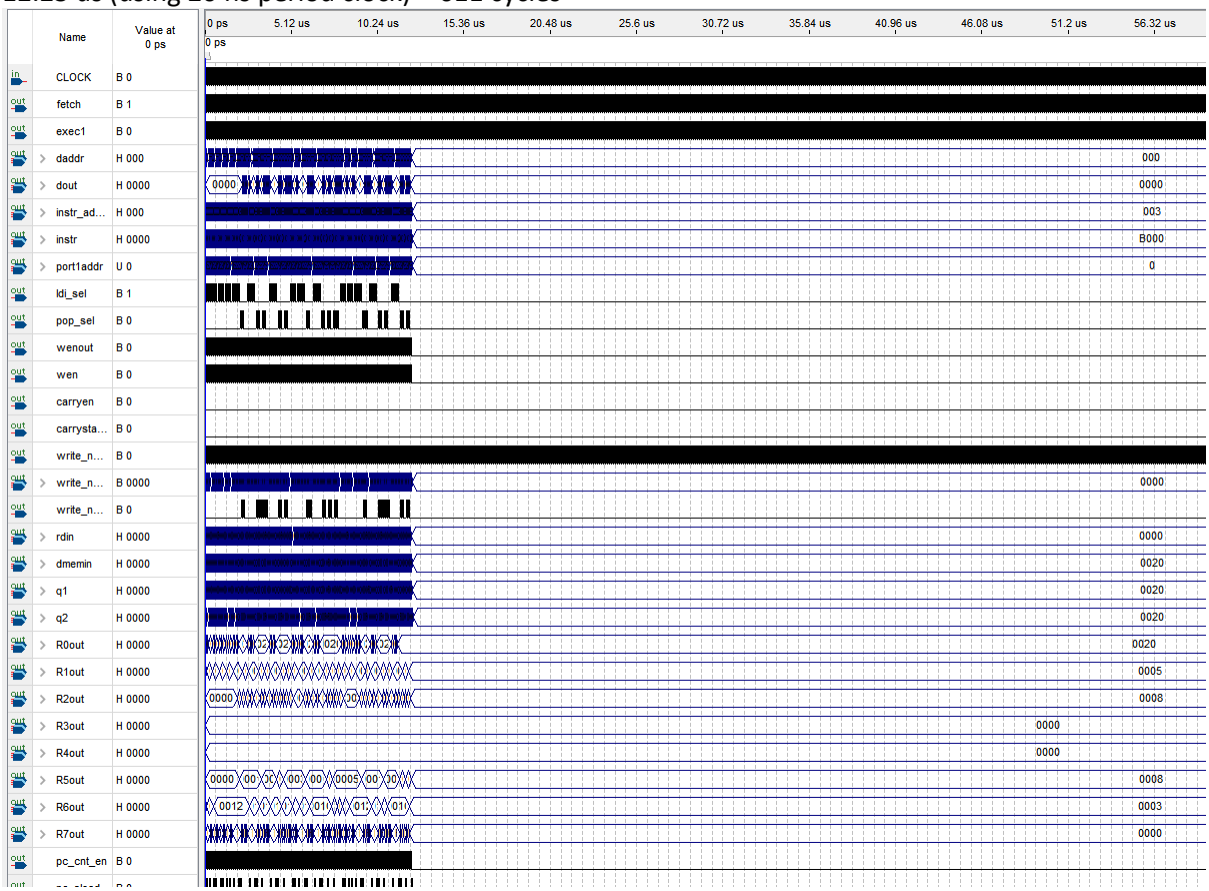
First section of Fib(2)



The stack pointer R7 could clearly be seen decrementing as values are pushed into the stack in the screenshot above. R6 also updates as the link register.

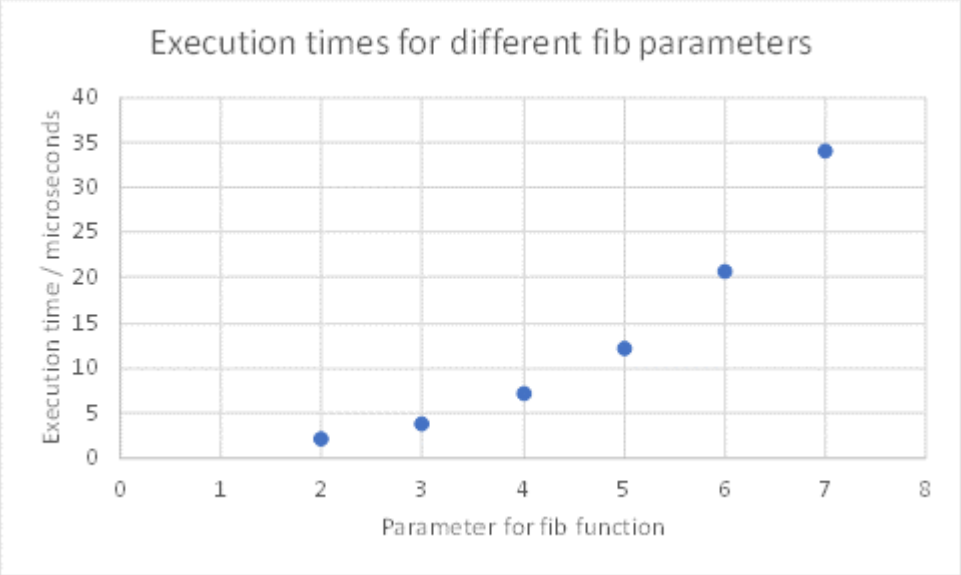
FIB(5)=8

12.23 us (using 20 ns period clock) = 611 cycles



See Appendix 10 for Fib simulations from fib(2) to fib(7).

Relationship between execution time (using 20 ns period clock) and parameter

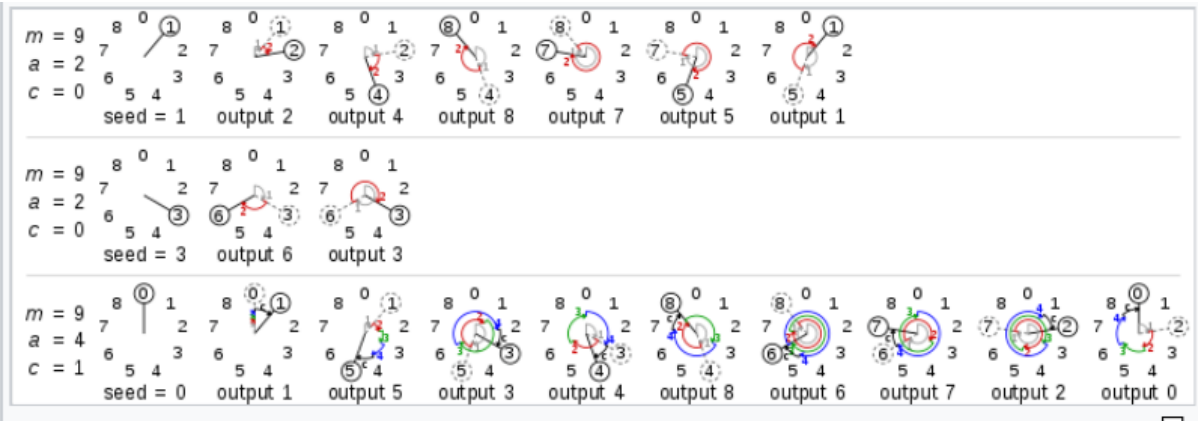


As the parameter was increased, the execution time increased exponentially. This was due to the use of a recursive implementation of the Fibonacci function, which required even more calls to itself as the parameter increases. This was an especially inefficient implementation of the function, and it would be interesting to see how a different implementation could be used to decrease execution time.

7.2 Benchmark: Pseudo-random integers with LCG

$$x_{n+1} = (ax_n + b) \bmod 2^N$$

Using linear congruential generator is one of the ways to generate pseudo- random integer, it takes three parameters a,b,N and one seed value Xn to generate the next value. The sequence generated by the generator is called a linear congruential sequence. The target of this research is to choose the suitable parameters that generate the longest sequence without repeating, so the sequence looks random. Figure LCG shows three examples with fixed sequence cycle.



Wikipedia diagram[7]

Firstly, the generator took a seed input which means the output is related with the input, hence the largest possible length of the sequence is 2^N. In this case, N was fixed at 16 as it is the word length. Ideally the maximum length of the sequence is 65536.

The first case is b= 0, a is a primitive element of 2^16, and assume 2^N is a prime number. The length of the sequence without repeating would be 2^16-1=65535. This special case is called “Lehmer random number generator”. However, 2^N is not a prime number, so this is not a possible case. When b=0, 2^N is a power of 2, these parameters are commonly used because it is convenient for binary representation. This form has maximum sequence length of 2^N/4 when a=3 or a=5 and the initial seed input Xn is odd. The final case is when b does not equal zero.

[7] Wikipedia, Linear Congruential generator.[online] available at: <https://en.wikipedia.org/wiki/Linear_congruential_generator>

According to Hull-Dobell Theorem[8], if

1. b and 2^N are relative prime,
2. a-1 is divisible by all prime factor of 2^N
3. a-1 is divisible by 4 if 2^N is divisible by 4

The period of the sequence is equal to 2^N.

Therefore, for this implementation if b=1, a=2^15+1=32769, N=16, in theory the sequence would have a length of 65536, which is much too large for the execution time of the Quartus simulation

Assembly:

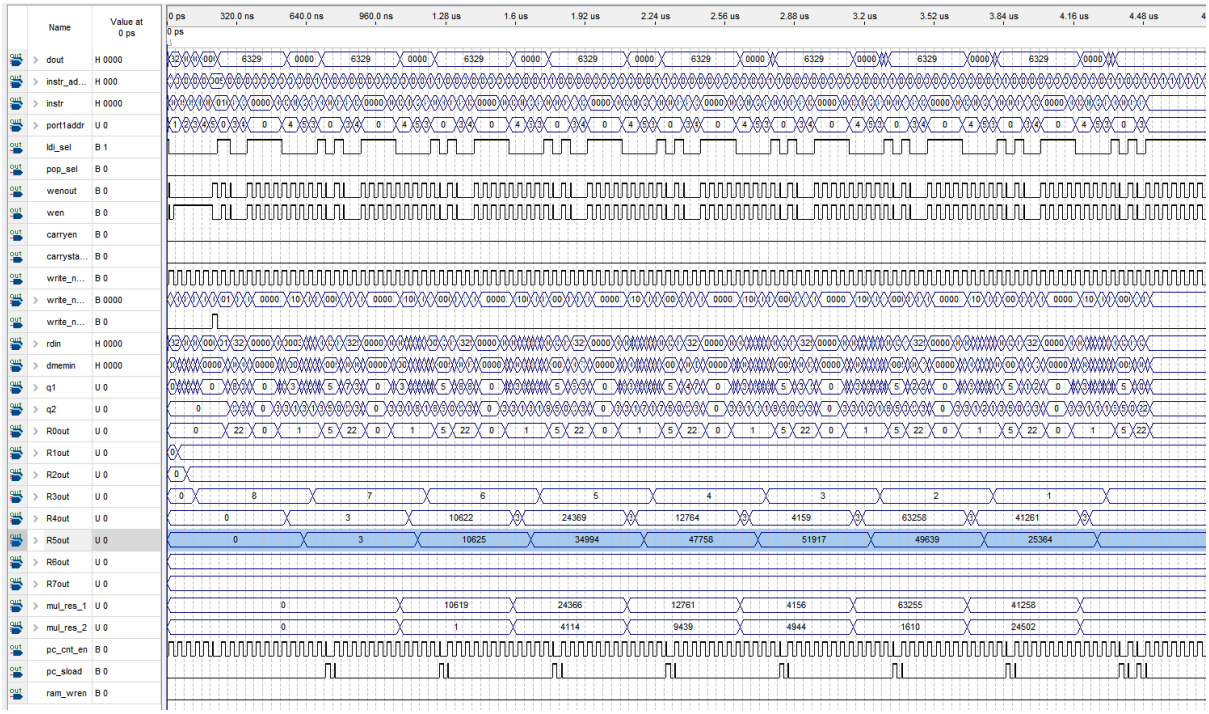
Pseudo Random Integer						
	General Instruction		OPCODE	OPERAND	Address	Hex
R1 = a = 25385	LDR 0 0 R1 R0		0001	000000001000	0	1008
R2 = b = 3	LDR 0 1 R2 R0		0001	000001010000	1	1050
R3 = n = 8	LDR 0 2 R3 R0		0001	000010011000	2	1098
R4 = s = 0 = y	LDR 0 3 R4 R0		0001	000011100000	3	10E0
R5 = sum = 0	LDR 0 3 R5 R0		0001	000011101000	4	10E8
LOOP	LDI END's address		0000	012	5	0012
	JEQ 0 R3 R0		0100	010000011000	6	4418
	MUL R4 R1		0111	000000100001	7	7021
Wait for MUL to finish	LDI 0		0000	000000000000	8	0000
	LDI 0		0000	000000000000	9	0000
	LDI 0		0000	000000000000	10	0000
	LDI 1		0000	000000000001	11	0001
	MOV 0 0 0 R4 R2		0011	000000100010	12	3022
Move LSB of product to	MOV 0 0 3 R4 0		0011	000011100000	13	30E0
R4	ADD R5 R4		0101	000000101100	14	502C
	SUB R3 R0		0110	010000011000	15	6418
	LDI Loop's address		0000	000000000005	16	0005
	JMP R0		0100	000000000000	17	4000
END	STP		1011	000000000000	18	B000
				Data	0	6329
					1	0003
					2	0008
					3	0000

[8] Linear Congruential Generator I section two, Cornell Department of Mathematics. [online] available at: <http://pi.math.cornell.edu/~mec/Winter2009/Luo/Linear%20Congruential%20Generator/linear%20congruential%20gen1.html>

The value stored in R0 is initialised with 0. Firstly, load R1 with 25385 as the "a" value, load R2 with 3 as the "b" value, load R3 with 8 as the "n" value. R4 stores "y" value, R5 stores "sum" value. "Y" and "sum" are initialised with 0 as well. Then write the stop address to R0, compare the value 0 and the “n” value stores in R3, if they are equal, jump to the END ‘address otherwise R4 multiplies with R1, the “y” and the “a” value. During the multiplication, write value 1 to R0. Once the multiplication is done, the least significant 16 bits have been stored in R4. Then the value stores in R5 is added with R4 which represent adding y value to the sum. After that, the n value is subtracted by 1 and jump back to the start of the loop where the comparison between n value and 0 has been done. The loop continues and at the end of each loop n value is subtracted by 1. Once the n value is one, the conditional jump will jump to the stop address and the test code for pseudo random integer has finished. The final random integer is stored at R5.

Testing:

Using "typical" parameters in overview
A=25385=0x6329, b=3, n=8
213 cycles
Execution time: 4.27 us
Pattern: 0, 3, 10625, 34994, 47758, 51917, 49639, 25364, 39500 (does not seem to repeat)



The outputs were changed to the radix unsigned integer to read the values more easily.

Even when increasing the length of the loop, the integers never seemed to repeat due to the small execution time of the Quartus simulation.

Execution time scaled linearly as the length of the for loop increased, as the for loop just executed the corresponding number of times, where each loop required the same amount of time.

7.3 Benchmark: Traverse linked list to find an item

Assembly:

Detail	Instructions	Opcode	Oprand	Address	Hex
LDI value to be found	LDI 007	0000	7	0	0007
R1 = x	MOV R1 R0	0011	000000001000	1	3008
LDI address of LOOP	LDI 007	0000	7	2	0007
R3 = address of LOOP	MOV R3 R0	0011	000000011000	3	3018
LDI address of END	LDI 00D	0000	D	4	000D
R4 = address of END	MOV R4 R0	0011	000000100000	5	3020
R0 = head	LDI head	0000	1	6	0001
LOOP: R2 = value in the linked list	LDR R2 R0	0001	000000010000	7	1010
R2 = R2 - x	SUB R2 R1	0110	010000010001	8	6411
Checks if head->value == x	JEQ 0 R2 R4	0100	010000010100	9	4414
Load the next address of node	LDR R0 [R0+1]	0001	000001000000	10	1040
Jumps if end of list reached	JEQ 0 R0 R4	0100	010000000100	11	4404
	JMP R3	0100	000000000011	12	4003
END	MOV R5 R0	0011	000000101000	13	3028
	STP	1011		14	B000

Note that to change the x value (value being searched for in the list), the first instruction needed to be changed to this x value. The above code looked for the number 7 in the list.

First the target value is loaded in R0 and stored in R1 in the next instruction, then load R0 with the beginning address of the loop and stores the address in R3 in the next instruction. After that load R0 with the end address of the loop and store the end address in R4 in the next instruction. After all the initialisation, R0 is loaded with the head pointer which contain the address of the first value, the value is compared with the target value by subtraction method. If it is the target value, jump to the end address and stored the pointer in R5, the test is finished. If it is not the target value, loading R0 with pointer pointing to the next value in the list and jump back to the beginning of the loop, this loop will stop when it iterates through the whole list or when the target value has been found. NULL was defined as the memory

address 0. The item was found if R5 (register 5) returns the address of the item with value x. If it remains at 0, the item was not found in the list.

Data memory:

Data	0	NULL	23	A
	1	1	24	19
	2	3	25	B
	3	2	26	1B
	4	6	27	C
	5	0	28	1D
	6	3	29	D
	7	C	30	1F
	8	0	31	E
	9	5	32	21
	10	F	33	F
	11	0	34	23
	12	4	35	10
	13	9	36	25
	14	0	37	11
	15	6	38	27
	16	11	39	12
	17	7	40	29
	18	13	41	13
	19	8	42	2B
	20	15	43	14
	21	9	44	0
	22	17		

This test data was used for the simulations (the list continues on the right side). One item in the linked list consisted of the value of the item and the value in the next memory address is the memory address pointing to the next item in the linked list (each item's value was highlighted in the image). The memory location storing a memory address acted like a sort of pointer. In this example, memory address 1 is the head item, which had a value of 1 and the next item in the list is at the memory address 3 (as the value in memory address 1+1 is 3). The length of this list was 20 items, although the actual length of the simulation depended on what value was being searched for in the list, as once it was found, the simulation stopped. The length was tested for 7, 10, 15, and 20 items (all simulation results for "find in list" benchmark found in Appendix 12).

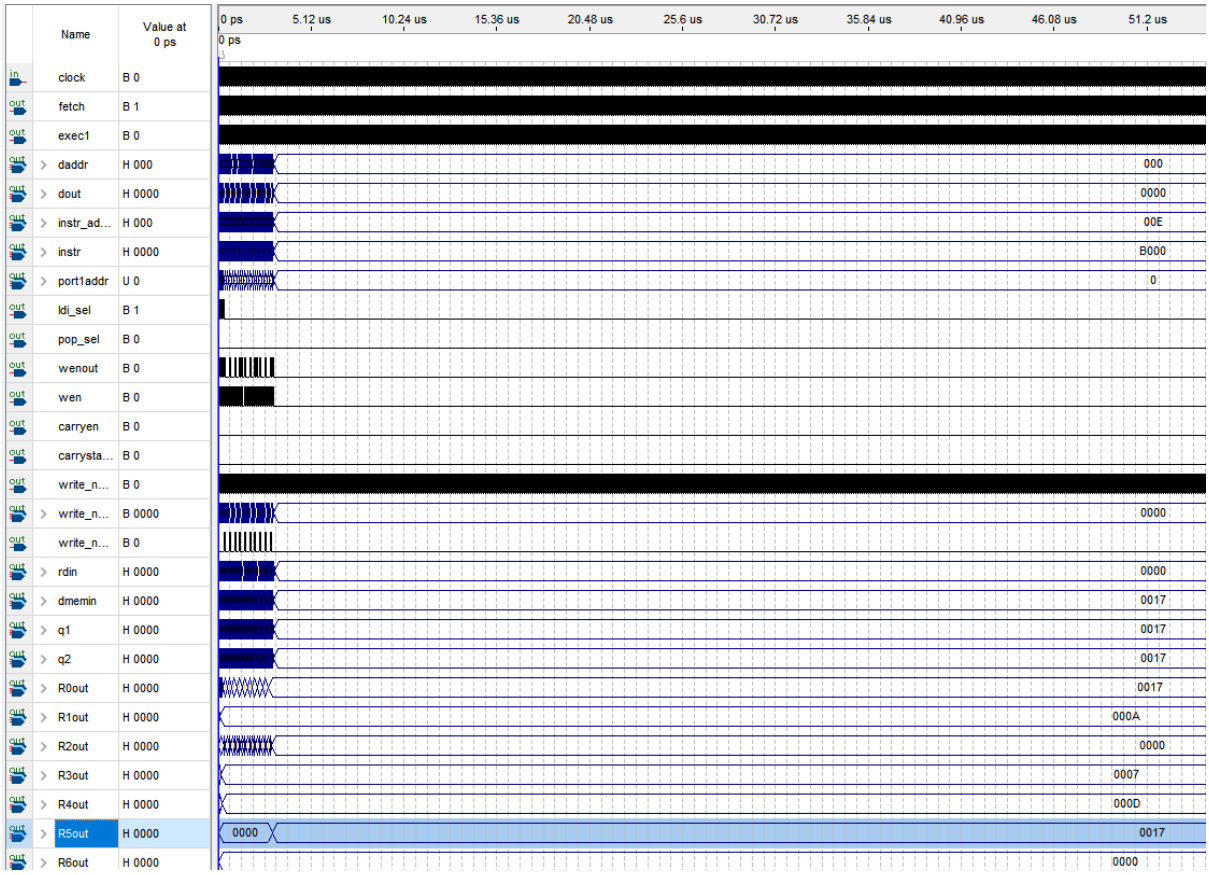
Testing:

Length 10 (typical):

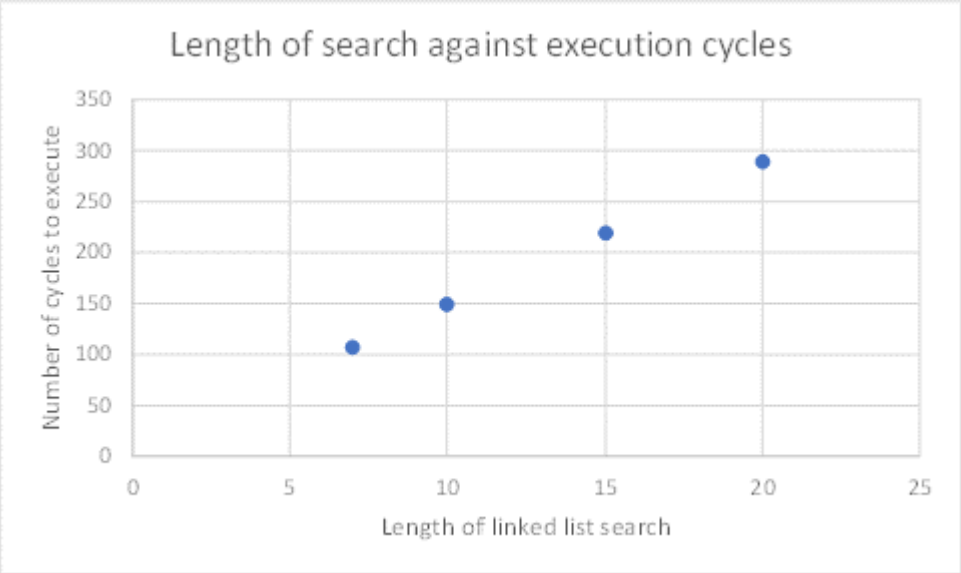
To test a length 10 linked list, the first instruction needed to be changed to 000A, as the linked list values are in numerical order, and thus it would stop at 10 items when 0xA or 10 is found as the value in the tenth item.

149 cycles

2.99 us



Relationship between execution time and linked list size



Execution time seemed to increase linearly with linked list size as each item took up the same amount of time. It would be interesting to investigate quicker search algorithms, possibly using binary search trees.

8 Optimisation

8.1 Basic CPU evaluation

The speed, power and area were measured using the full compilation on Quartus. This evaluation is for the unpipelined CPU.

	Fmax	Restricted Fmax	Clock Name
1	60.65 MHz	60.65 MHz	clock

To find the geometric mean time, a parameter was fixed for each benchmark tests that obtained similar execution times between each test. This allowed comparison between geometric mean times.

FIB(3) = 3
191 cycles = 3.149 us

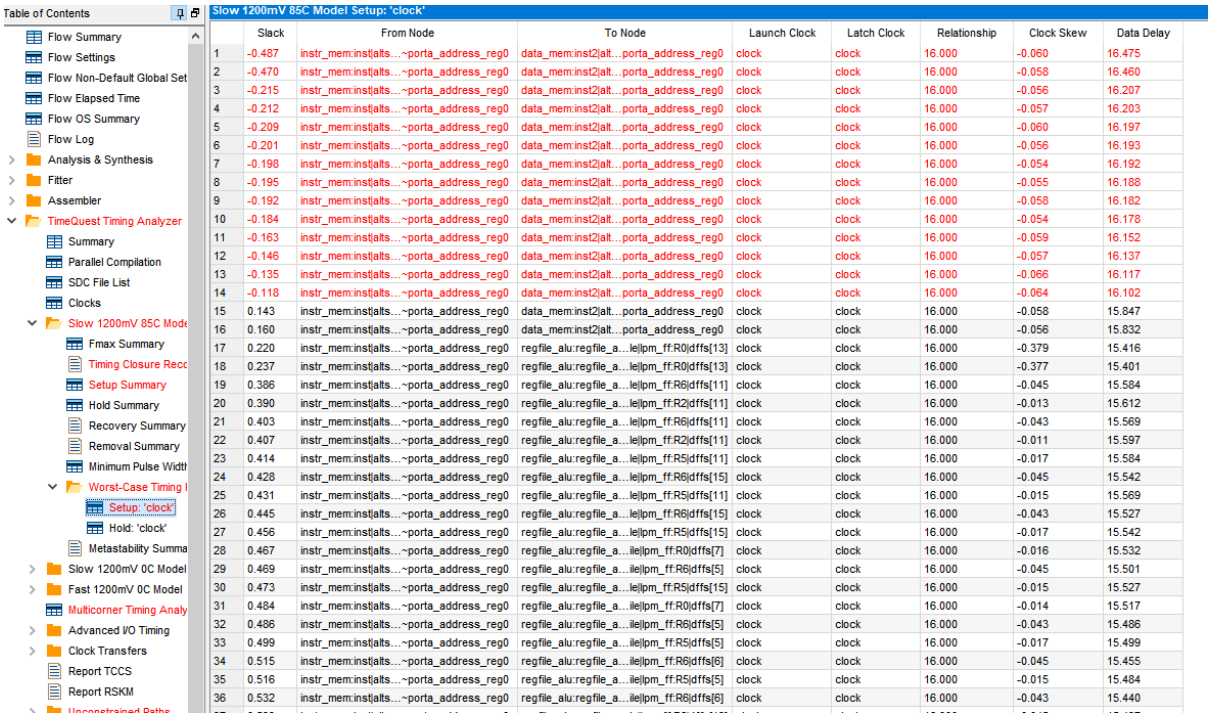
LCG typical parameters A=25385=0x6329, b=3, n=8
213 cycles = 3.512 us

Length 15 linked list:
219 cycles = 3.611 us

Geometric mean time = 3.42 us

Analysis of CPU block:

Timing analysis:

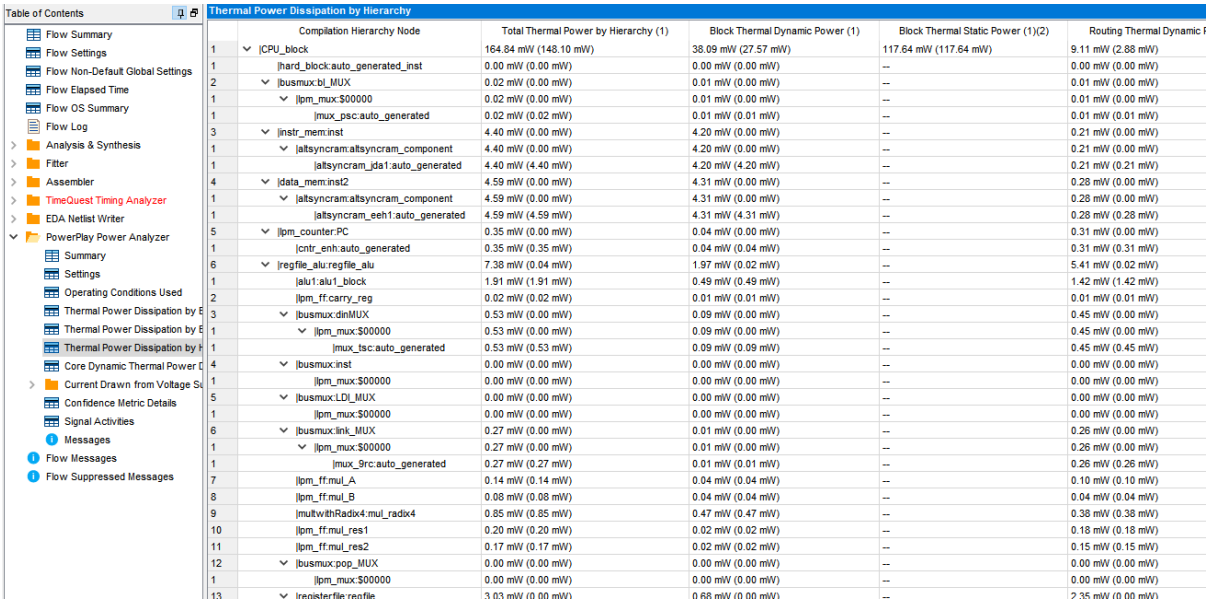


	Slack	From Node	To Node	Launch Clock	Latch Clock	Relationship	Clock Skew	Data Delay
1	-0.487	instr_mem:inst|alts...~porta_address_reg0	data_mem:inst2|alt...porta_address_reg0	clock	clock	16.000	-0.060	16.475
2	-0.470	instr_mem:inst|alts...~porta_address_reg0	data_mem:inst2|alt...porta_address_reg0	clock	clock	16.000	-0.058	16.460
3	-0.215	instr_mem:inst|alts...~porta_address_reg0	data_mem:inst2|alt...porta_address_reg0	clock	clock	16.000	-0.056	16.207
4	-0.212	instr_mem:inst|alts...~porta_address_reg0	data_mem:inst2|alt...porta_address_reg0	clock	clock	16.000	-0.057	16.203
5	-0.209	instr_mem:inst|alts...~porta_address_reg0	data_mem:inst2|alt...porta_address_reg0	clock	clock	16.000	-0.060	16.197
6	-0.201	instr_mem:inst|alts...~porta_address_reg0	data_mem:inst2|alt...porta_address_reg0	clock	clock	16.000	-0.056	16.193
7	-0.198	instr_mem:inst|alts...~porta_address_reg0	data_mem:inst2|alt...porta_address_reg0	clock	clock	16.000	-0.054	16.192
8	-0.195	instr_mem:inst|alts...~porta_address_reg0	data_mem:inst2|alt...porta_address_reg0	clock	clock	16.000	-0.055	16.188
9	-0.192	instr_mem:inst|alts...~porta_address_reg0	data_mem:inst2|alt...porta_address_reg0	clock	clock	16.000	-0.058	16.182
10	-0.184	instr_mem:inst|alts...~porta_address_reg0	data_mem:inst2|alt...porta_address_reg0	clock	clock	16.000	-0.054	16.178
11	-0.163	instr_mem:inst|alts...~porta_address_reg0	data_mem:inst2|alt...porta_address_reg0	clock	clock	16.000	-0.059	16.152
12	-0.146	instr_mem:inst|alts...~porta_address_reg0	data_mem:inst2|alt...porta_address_reg0	clock	clock	16.000	-0.057	16.137
13	-0.135	instr_mem:inst|alts...~porta_address_reg0	data_mem:inst2|alt...porta_address_reg0	clock	clock	16.000	-0.066	16.117
14	-0.118	instr_mem:inst|alts...~porta_address_reg0	data_mem:inst2|alt...porta_address_reg0	clock	clock	16.000	-0.064	16.102
15	0.143	instr_mem:inst|alts...~porta_address_reg0	data_mem:inst2|alt...porta_address_reg0	clock	clock	16.000	-0.058	15.847
16	0.160	instr_mem:inst|alts...~porta_address_reg0	data_mem:inst2|alt...porta_address_reg0	clock	clock	16.000	-0.056	15.832
17	0.220	instr_mem:inst|alts...~porta_address_reg0	regfile_alu:regfile_a...le|lpm_ff:R0|dffs[13]	clock	clock	16.000	-0.379	15.416
18	0.237	instr_mem:inst|alts...~porta_address_reg0	regfile_alu:regfile_a...le|lpm_ff:R0|dffs[13]	clock	clock	16.000	-0.377	15.401
19	0.386	instr_mem:inst|alts...~porta_address_reg0	regfile_alu:regfile_a...le|lpm_ff:R6|dffs[11]	clock	clock	16.000	-0.045	15.584
20	0.390	instr_mem:inst|alts...~porta_address_reg0	regfile_alu:regfile_a...le|lpm_ff:R2|dffs[11]	clock	clock	16.000	-0.013	15.612
21	0.403	instr_mem:inst|alts...~porta_address_reg0	regfile_alu:regfile_a...le|lpm_ff:R6|dffs[11]	clock	clock	16.000	-0.043	15.569
22	0.407	instr_mem:inst|alts...~porta_address_reg0	regfile_alu:regfile_a...le|lpm_ff:R2|dffs[11]	clock	clock	16.000	-0.011	15.597
23	0.414	instr_mem:inst|alts...~porta_address_reg0	regfile_alu:regfile_a...le|lpm_ff:R5|dffs[11]	clock	clock	16.000	-0.017	15.584
24	0.428	instr_mem:inst|alts...~porta_address_reg0	regfile_alu:regfile_a...le|lpm_ff:R6|dffs[15]	clock	clock	16.000	-0.045	15.542
25	0.431	instr_mem:inst|alts...~porta_address_reg0	regfile_alu:regfile_a...le|lpm_ff:R5|dffs[11]	clock	clock	16.000	-0.015	15.569
26	0.445	instr_mem:inst|alts...~porta_address_reg0	regfile_alu:regfile_a...le|lpm_ff:R6|dffs[15]	clock	clock	16.000	-0.043	15.527
27	0.456	instr_mem:inst|alts...~porta_address_reg0	regfile_alu:regfile_a...le|lpm_ff:R5|dffs[15]	clock	clock	16.000	-0.017	15.542
28	0.467	instr_mem:inst|alts...~porta_address_reg0	regfile_alu:regfile_a...ile|lpm_ff:R0|dffs[7]	clock	clock	16.000	-0.016	15.532
29	0.469	instr_mem:inst|alts...~porta_address_reg0	regfile_alu:regfile_a...ile|lpm_ff:R6|dffs[5]	clock	clock	16.000	-0.045	15.501
30	0.473	instr_mem:inst|alts...~porta_address_reg0	regfile_alu:regfile_a...le|lpm_ff:R5|dffs[15]	clock	clock	16.000	-0.015	15.527
31	0.484	instr_mem:inst|alts...~porta_address_reg0	regfile_alu:regfile_a...ile|lpm_ff:R0|dffs[7]	clock	clock	16.000	-0.014	15.517
32	0.486	instr_mem:inst|alts...~porta_address_reg0	regfile_alu:regfile_a...ile|lpm_ff:R6|dffs[5]	clock	clock	16.000	-0.043	15.486
33	0.499	instr_mem:inst|alts...~porta_address_reg0	regfile_alu:regfile_a...ile|lpm_ff:R5|dffs[5]	clock	clock	16.000	-0.017	15.499
34	0.515	instr_mem:inst|alts...~porta_address_reg0	regfile_alu:regfile_a...ile|lpm_ff:R6|dffs[6]	clock	clock	16.000	-0.045	15.455
35	0.516	instr_mem:inst|alts...~porta_address_reg0	regfile_alu:regfile_a...ile|lpm_ff:R5|dffs[5]	clock	clock	16.000	-0.015	15.484
36	0.532	instr_mem:inst|alts...~porta_address_reg0	regfile_alu:regfile_a...ile|lpm_ff:R6|dffs[6]	clock	clock	16.000	-0.043	15.440

The largest propagation delay in the CPU_block bdf was between the instruction and data memories. In the CPU, there is a large propagation delay due to the entire loop being completed in one cycle and being required for the next cycle. One way to decrease this propagation delay is to add pipelining stages. A register could be added at the output of the data RAM so that the output is ready at the next cycle. This would involve changes to the state machine so that all the stages have a value in each register, and then this could be pipelined so that each stage has a very log propagation delay, thus increasing the max clock frequency (discussed further in Extension).

Power:

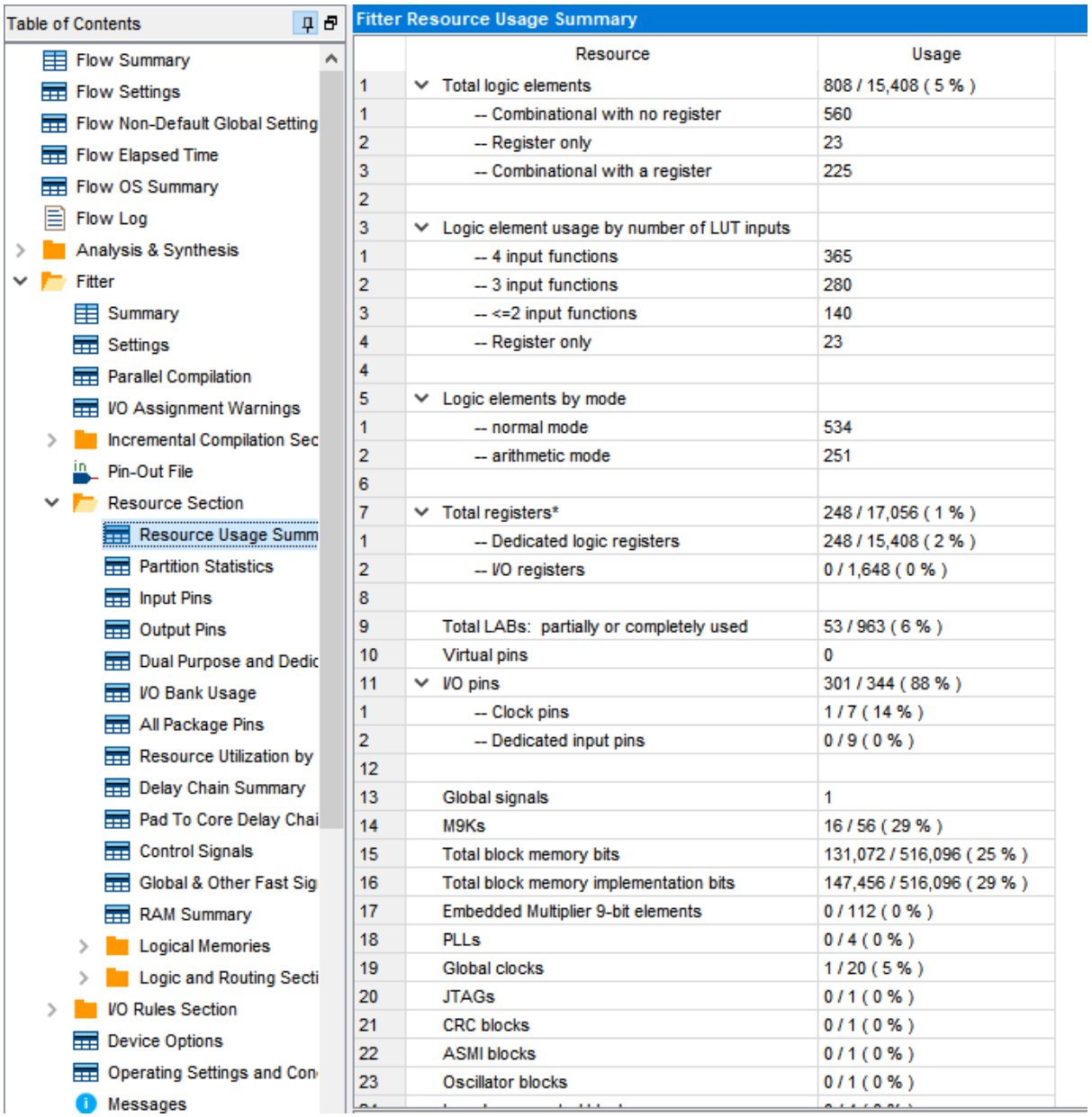
Total Thermal Power Dissipation	235.22 mW
Core Dynamic Thermal Power Dissipation	19.63 mW
Core Static Thermal Power Dissipation	49.78 mW
I/O Thermal Power Dissipation	165.81 mW



Thermal Power Dissipation by Hierarchy

	Compilation Hierarchy Node	Total Thermal Power by Hierarchy (1)	Block Thermal Dynamic Power (1)	Block Thermal Static Power (1)(2)	Routing Thermal Dynamic
1	|CPU_block	164.84 mW (148.10 mW)	38.09 mW (27.57 mW)	117.64 mW (117.64 mW)	9.11 mW (2.88 mW)
1	|hard_block:auto_generated_inst	0.00 mW (0.00 mW)	0.00 mW (0.00 mW)	--	0.00 mW (0.00 mW)
2	|busmux:bl_MUX	0.02 mW (0.00 mW)	0.01 mW (0.00 mW)	--	0.01 mW (0.00 mW)
1	|lpm_mux:$00000	0.02 mW (0.00 mW)	0.01 mW (0.00 mW)	--	0.01 mW (0.00 mW)
1	|mux_psc:auto_generated	0.02 mW (0.02 mW)	0.01 mW (0.01 mW)	--	0.01 mW (0.01 mW)
3	|instr_mem:inst	4.40 mW (0.00 mW)	4.20 mW (0.00 mW)	--	0.21 mW (0.00 mW)
1	|altsyncram:altsyncram_component	4.40 mW (0.00 mW)	4.20 mW (0.00 mW)	--	0.21 mW (0.00 mW)
1	|altsyncram_jda1:auto_generated	4.40 mW (4.40 mW)	4.20 mW (4.20 mW)	--	0.21 mW (0.21 mW)
4	|data_mem:inst2	4.59 mW (0.00 mW)	4.31 mW (0.00 mW)	--	0.28 mW (0.00 mW)
1	|altsyncram:altsyncram_component	4.59 mW (0.00 mW)	4.31 mW (0.00 mW)	--	0.28 mW (0.00 mW)
1	|altsyncram_eeh1:auto_generated	4.59 mW (4.59 mW)	4.31 mW (4.31 mW)	--	0.28 mW (0.28 mW)
5	|lpm_counter:PC	0.35 mW (0.00 mW)	0.04 mW (0.00 mW)	--	0.31 mW (0.00 mW)
1	|cntr_enh:auto_generated	0.35 mW (0.35 mW)	0.04 mW (0.04 mW)	--	0.31 mW (0.31 mW)
6	|regfile_alu:regfile_alu	7.38 mW (0.04 mW)	1.97 mW (0.02 mW)	--	5.41 mW (0.02 mW)
1	|alu1:alu1_block	1.91 mW (1.91 mW)	0.49 mW (0.49 mW)	--	1.42 mW (1.42 mW)
2	|lpm_ff:carry_reg	0.02 mW (0.02 mW)	0.01 mW (0.01 mW)	--	0.01 mW (0.01 mW)
3	|busmux:dinMUX	0.53 mW (0.00 mW)	0.09 mW (0.00 mW)	--	0.45 mW (0.00 mW)
1	|lpm_mux:$00000	0.53 mW (0.00 mW)	0.09 mW (0.00 mW)	--	0.45 mW (0.00 mW)
1	|mux_tsc:auto_generated	0.53 mW (0.53 mW)	0.09 mW (0.09 mW)	--	0.45 mW (0.45 mW)
4	|busmux:inst	0.00 mW (0.00 mW)	0.00 mW (0.00 mW)	--	0.00 mW (0.00 mW)
1	|lpm_mux:$00000	0.00 mW (0.00 mW)	0.00 mW (0.00 mW)	--	0.00 mW (0.00 mW)
5	|busmux:LDI_MUX	0.00 mW (0.00 mW)	0.00 mW (0.00 mW)	--	0.00 mW (0.00 mW)
1	|lpm_mux:$00000	0.00 mW (0.00 mW)	0.00 mW (0.00 mW)	--	0.00 mW (0.00 mW)
6	|busmux:link_MUX	0.27 mW (0.00 mW)	0.01 mW (0.00 mW)	--	0.26 mW (0.00 mW)
1	|lpm_mux:$00000	0.27 mW (0.00 mW)	0.01 mW (0.00 mW)	--	0.26 mW (0.00 mW)
1	|mux_9rc:auto_generated	0.27 mW (0.27 mW)	0.01 mW (0.01 mW)	--	0.26 mW (0.26 mW)
7	|lpm_ff:mul_A	0.14 mW (0.14 mW)	0.04 mW (0.04 mW)	--	0.10 mW (0.10 mW)
8	|lpm_ff:mul_B	0.08 mW (0.08 mW)	0.04 mW (0.04 mW)	--	0.04 mW (0.04 mW)
9	|multwithRadix4:mul_radix4	0.85 mW (0.85 mW)	0.47 mW (0.47 mW)	--	0.38 mW (0.38 mW)
10	|lpm_ff:mul_res1	0.20 mW (0.20 mW)	0.02 mW (0.02 mW)	--	0.18 mW (0.18 mW)
11	|lpm_ff:mul_res2	0.17 mW (0.17 mW)	0.02 mW (0.02 mW)	--	0.15 mW (0.15 mW)
12	|busmux:pop_MUX	0.00 mW (0.00 mW)	0.00 mW (0.00 mW)	--	0.00 mW (0.00 mW)
1	|lpm_mux:$00000	0.00 mW (0.00 mW)	0.00 mW (0.00 mW)	--	0.00 mW (0.00 mW)
13	|registerfile:regfile	3.03 mW (0.00 mW)	0.68 mW (0.00 mW)	--	2.35 mW (0.00 mW)

Area:



Fitter Resource Usage Summary

	Resource	Usage
1	Total logic elements	808 / 15,408 (5 %)
1	-- Combinational with no register	560
2	-- Register only	23
3	-- Combinational with a register	225
2		
3	Logic element usage by number of LUT inputs	
1	-- 4 input functions	365
2	-- 3 input functions	280
3	-- <=2 input functions	140
4	-- Register only	23
4		
5	Logic elements by mode	
1	-- normal mode	534
2	-- arithmetic mode	251
6		
7	Total registers*	248 / 17,056 (1 %)
1	-- Dedicated logic registers	248 / 15,408 (2 %)
2	-- I/O registers	0 / 1,648 (0 %)
8		
9	Total LABs: partially or completely used	53 / 963 (6 %)
10	Virtual pins	0
11	I/O pins	301 / 344 (88 %)
1	-- Clock pins	1 / 7 (14 %)
2	-- Dedicated input pins	0 / 9 (0 %)
12		
13	Global signals	1
14	M9Ks	16 / 56 (29 %)
15	Total block memory bits	131,072 / 516,096 (25 %)
16	Total block memory implementation bits	147,456 / 516,096 (29 %)
17	Embedded Multiplier 9-bit elements	0 / 112 (0 %)
18	PLLs	0 / 4 (0 %)
19	Global clocks	1 / 20 (5 %)
20	JTAGs	0 / 1 (0 %)
21	CRC blocks	0 / 1 (0 %)
22	ASMI blocks	0 / 1 (0 %)
23	Oscillator blocks	0 / 1 (0 %)

Power and area were more difficult to change than speed, although they were taken into account when choosing the multiplier to carry out multiplication.

8.2 Choosing the multiplier

To maximise the clock frequency of the CPU, the propagation delay needed to be minimised. Therefore, the power and latency analysis were done with the Radix 2 multiplier (Figure analysis Radix 2 Appendix 8) and the Radix 4 multiplier(Figure analysis Radix 4 Appendix 7). After running the full compilation with model Cyclone IV E, the TimeQuest Timing Analyzer showed the time analysis.

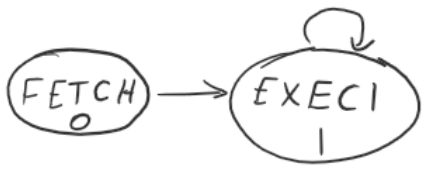
The set-up slack in time analysis means the difference between the data required time and the data arrival time. For the radix 4 multiplier, the worst-case slack is 0.395, for the radix 2 multiplier is 0.677. The radix 4 had half the cycles, 0.395*2>0.677. The result showed that the propagation delay for radix 4 multiplier was less compared to radix 2. The positive slack in each simulation meant there were still some margin for both multipliers to increase the clock frequency further.

The dynamic power is the power consumed when inputs are active, the static power is the power consumption the inputs are kept constant, it is usually caused by DFF. For the Radix 2 multiplier, total power dissipation was 97.79mV, core dynamic thermal power dissipation was 6.19mW, and core static thermal power dissipation was 42.86mW. For Radix 4 multiplier, total power dissipation was 102.71mW, core dynamic thermal power dissipation was 7.42mW, and core static thermal power dissipation was 42.88mW. The result makes perfect sense as the high dynamic power dissipation for radix 4 was because it shifted more bits in the same amount of time. The similar static power was because the same number of additions had been done in the same amount of time. Even the Radix 4 multiplier had a slightly larger total power dissipation, however it could do double the work that radix could do with the same amount of time. Hence, the radix 4 multiplier was the most efficient choice.

8.3 Pipelining

Pipelining[9] involved creating a state machine that fetched the next instruction and executed the current instruction during the same cycle. This approximately halved the number of cycles required as fetch and exec1 happen at the same time.

Pipelined state machine:

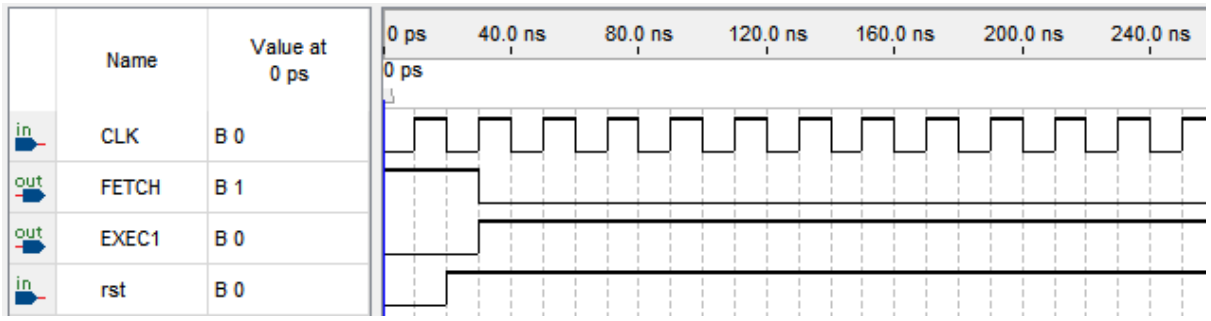


[9] Torsten Grust, Class Lecture, Topic "*Pipelining*" Database Systems and Modern CPU Architecture, Eberhard Karls University of Tübingen, German, 2009. Available at: <https://db.inf.uni-tuebingen.de/staticfiles/teaching/ss09/dbcpu/dbms-cpu-2.pdf>

The new state machine executes FETCH only for the first cycle and then fetches the next instruction while executing the current instruction during EXEC1.

```
1   module pipeline_state_machine
2   (
3       input S,
4       output NS,
5       output FETCH,
6       output EXEC1
7   );
8
9   assign NS = 1;
10
11  assign FETCH = ~S;
12  assign EXEC1 = S;
13
14  endmodule
15
```

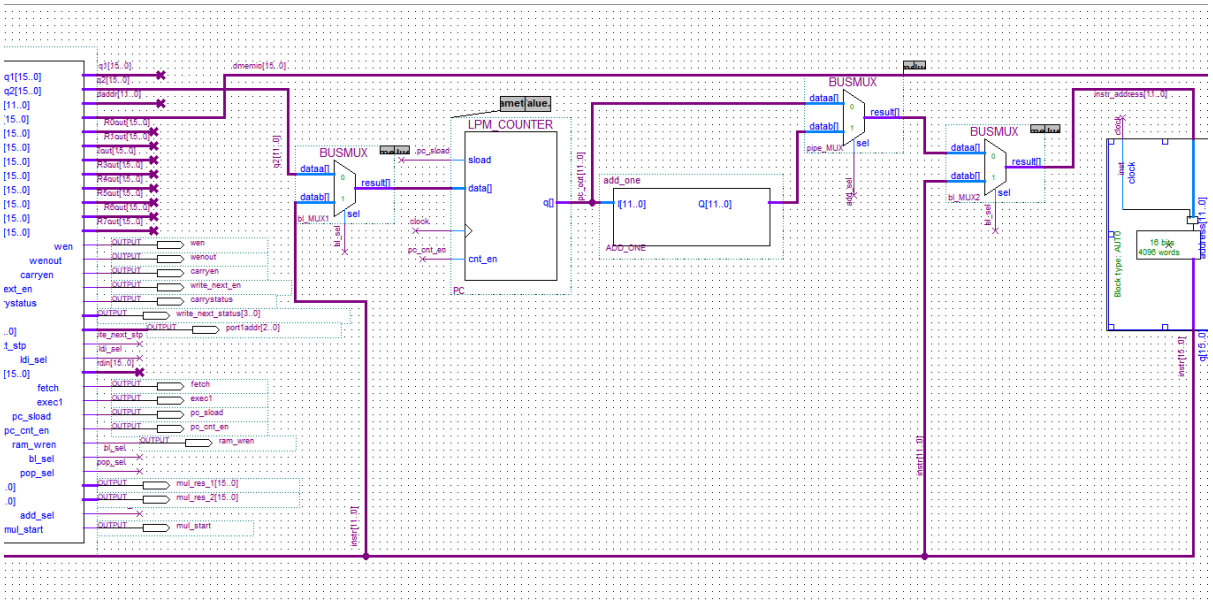
A reset pin was required for the DFF, so that FETCH starts at 1 at the beginning.



Pipeline hardware (see Appendix 2):

To increase PC during the first cycle, an add_one block was used for all instructions (adds one to output of PC). A MUX chose between PC+1 and PC depending on add_sel, so it should always increase by 1 other than for stop instructions, where it should not increase again. Otherwise, the PC would count above the STP instruction and skip it.

Hardware:



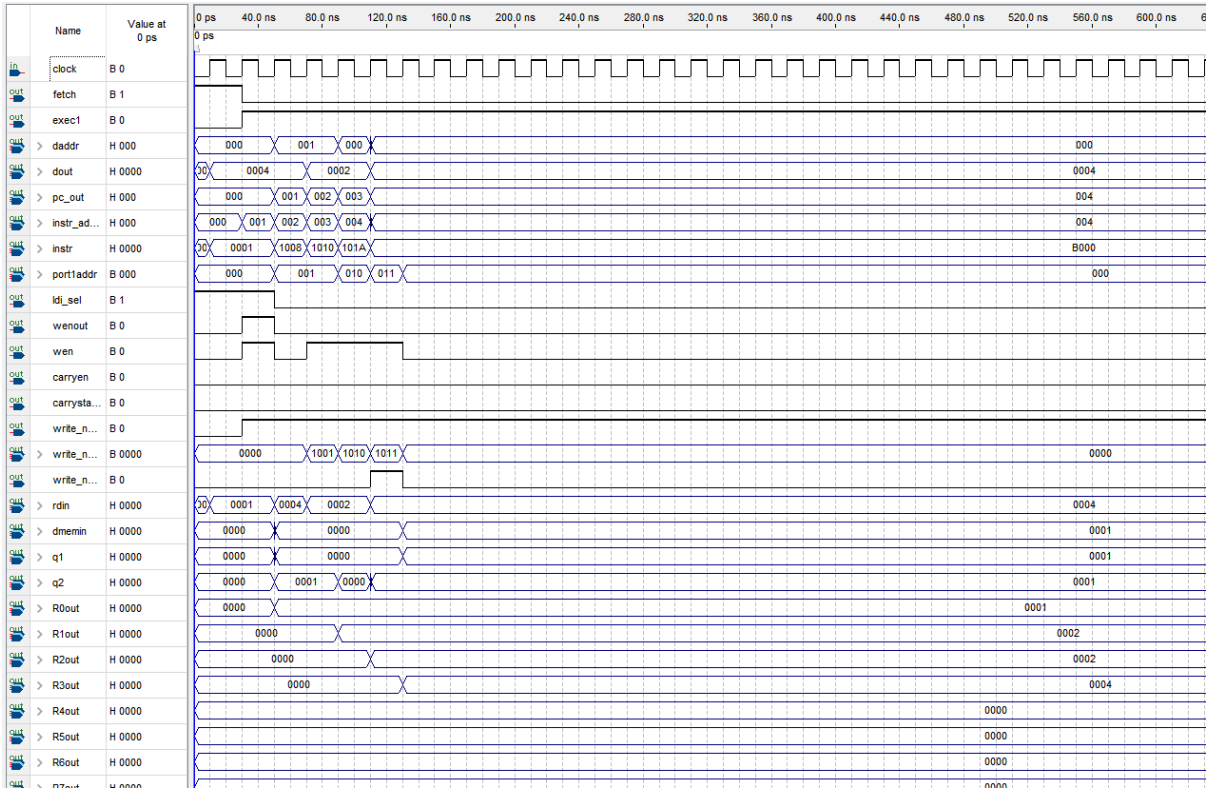
Pipelined control signals logic

```
// PC and RAM control signals
assign pc_sload = exec1 & (jmp_cond | bl);
assign pc_cnt_en = exec1 & ~(jmp_cond | stp | bl | write_next_stp);
assign ram_wren = exec1 & (str | stmfd);
assign add_sel = exec1 & ~(stp | jmp_cond | bl | write_next_stp);
```

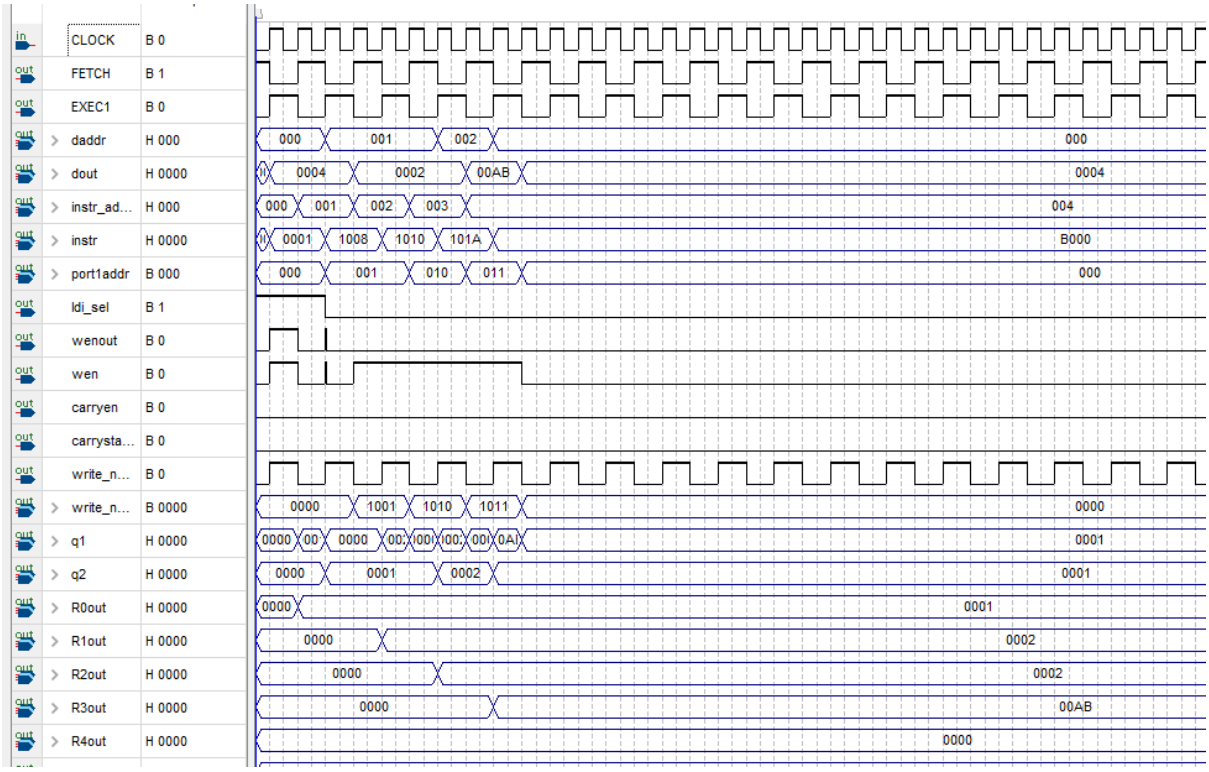
For pipelining, JMP and BL should not increase the PC, and they should not use PC+1, as this skips the address being jumped to. Write_next_stp (the delay for write instructions after an LDR) also had the same logic as it should stop the PC from counting during this cycle. BL also required an additional MUX at the input of the instruction memory so that the jump address updates during the same cycle as the BL instruction being executed, so that the next instruction already executes the instruction that was jumped to as it should be available at the memory output.

Improved Multiple LDR instructions Test:

Pipelined:



Unpipelined:



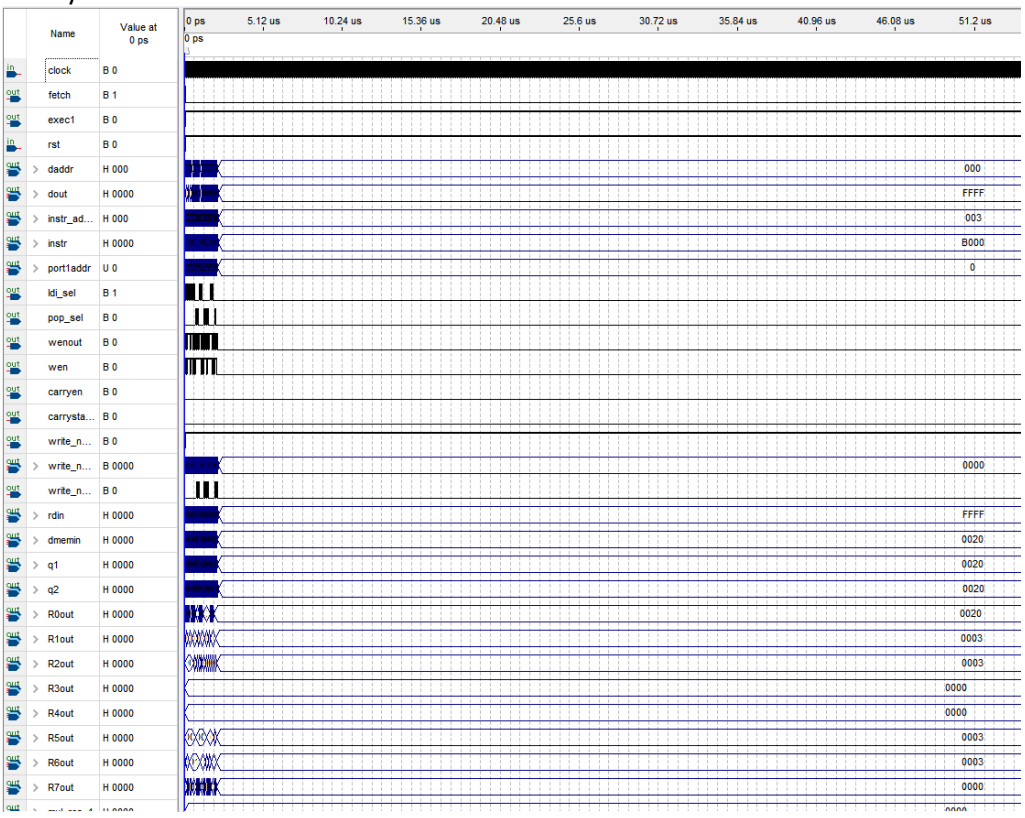
Clearly the pipelined version was able to execute the LDR instructions in one cycle rather than two cycles for the unpipelined version. This test was already optimised due to the LDR implementation allowing for LDR instructions to be carried out in parallel, but the pipelining allowed for even better optimisation in terms of number of cycles.

Multiplication while pipelining

More instructions had to be added after MUL instruction so that 9 instructions or cycles pass to give the multiplier time to obtain the output. This could not be decreased using pipelining, which meant that the LCG benchmark (the one using multiplication) did not have as great an improvement in execution time compared to the other benchmarks, as the majority of the time was taken up by the multiplication. While the ability to carry out instructions in parallel was possible, this could not be fully taken advantage of for this specific benchmark as not many instructions could be executed in parallel that would actually lead to improved efficiency.

Pipelined CPU benchmark tests (using fixed parameters for comparison)

Pipelined: FIB(3) = 3
106 cycles



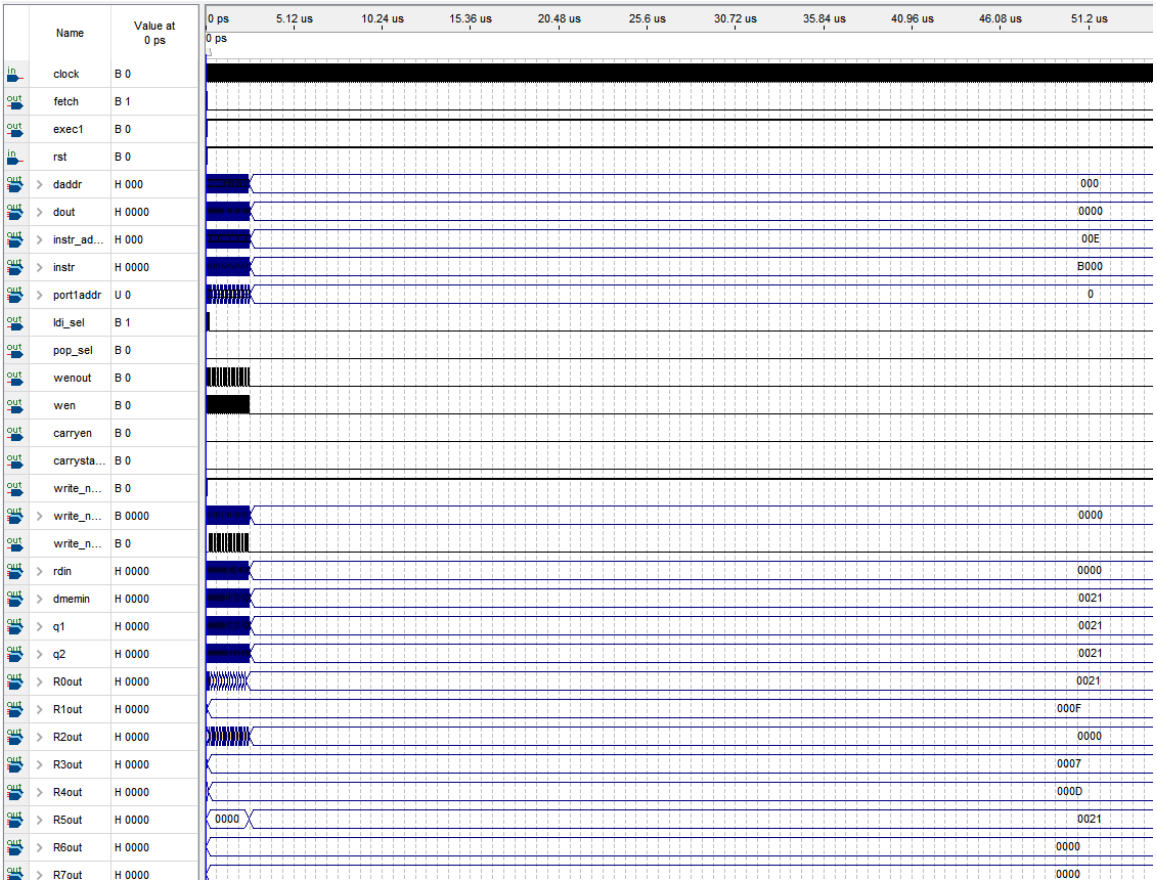
Pipelined: LCG with typical parameters:

A=25385=0x6329, b=3, n=8

163 cycles

Pipelined: Length 15 linked list search:

126 cycles



Each of the tests were clearly executed in less cycles. It was approximately halved from the unpipelined version, other than the LCG due to multiplication causing the same delay.

Pipelined compilation

Slow 1200mV 85C Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name	Note
1	63.31 MHz	63.31 MHz	clock	

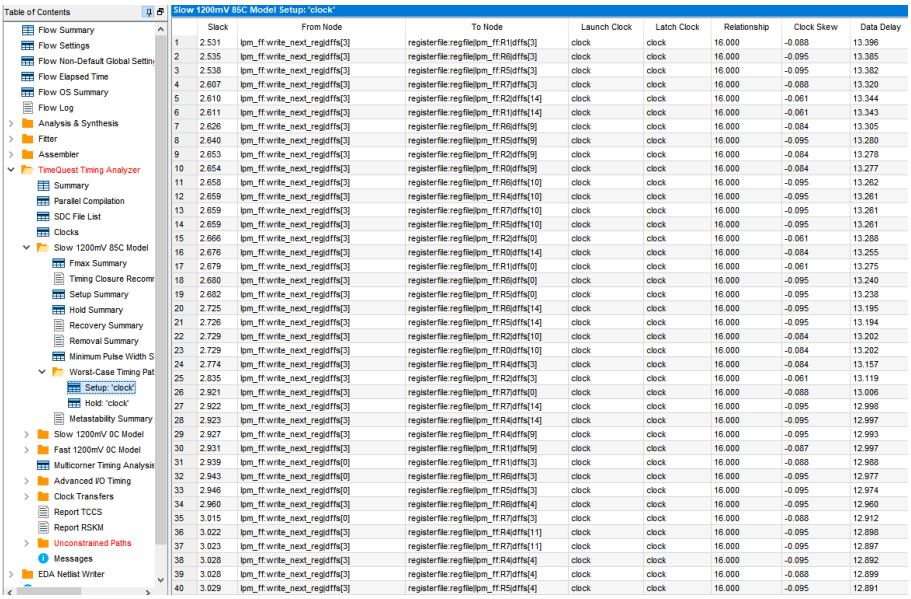
Fmax increased compared to the unpipelined CPU from 60 MHz to 63 MHz.

Geometric Mean Time (pipelined) = 2.047 microseconds

This was an increase from 3.42 microseconds. It is not exactly halved due to the long delay that remains due to the multiplier block of 9 cycles. This took up the majority of the execution time for the LCG benchmark.

Reg_Alu compilation :

By compiling this bdf, the paths that have the greatest propagation delay could be found.



	Slack	From Node	To Node	Launch Clock	Latch Clock	Relationship	Clock Skew	Data Delay
1	2.531	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R1|dffs[3]	clock	clock	16.000	-0.088	13.396
2	2.535	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R6|dffs[3]	clock	clock	16.000	-0.095	13.385
3	2.538	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R5|dffs[3]	clock	clock	16.000	-0.095	13.382
4	2.607	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R7|dffs[3]	clock	clock	16.000	-0.088	13.320
5	2.610	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R2|dffs[14]	clock	clock	16.000	-0.061	13.344
6	2.611	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R1|dffs[14]	clock	clock	16.000	-0.061	13.343
7	2.626	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R6|dffs[9]	clock	clock	16.000	-0.084	13.305
8	2.640	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R5|dffs[9]	clock	clock	16.000	-0.095	13.280
9	2.653	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R2|dffs[9]	clock	clock	16.000	-0.084	13.278
10	2.654	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R0|dffs[9]	clock	clock	16.000	-0.084	13.277
11	2.658	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R6|dffs[10]	clock	clock	16.000	-0.095	13.262
12	2.659	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R4|dffs[10]	clock	clock	16.000	-0.095	13.261
13	2.659	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R7|dffs[10]	clock	clock	16.000	-0.095	13.261
14	2.659	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R5|dffs[10]	clock	clock	16.000	-0.095	13.261
15	2.666	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R2|dffs[0]	clock	clock	16.000	-0.061	13.288
16	2.676	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R0|dffs[14]	clock	clock	16.000	-0.084	13.255
17	2.679	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R1|dffs[0]	clock	clock	16.000	-0.061	13.275
18	2.680	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R6|dffs[0]	clock	clock	16.000	-0.095	13.240
19	2.682	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R5|dffs[0]	clock	clock	16.000	-0.095	13.238
20	2.725	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R6|dffs[14]	clock	clock	16.000	-0.095	13.195
21	2.726	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R5|dffs[14]	clock	clock	16.000	-0.095	13.194
22	2.729	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R2|dffs[10]	clock	clock	16.000	-0.084	13.202
23	2.729	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R0|dffs[10]	clock	clock	16.000	-0.084	13.202
24	2.774	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R4|dffs[3]	clock	clock	16.000	-0.084	13.157
25	2.835	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R2|dffs[3]	clock	clock	16.000	-0.061	13.119
26	2.921	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R7|dffs[0]	clock	clock	16.000	-0.088	13.006
27	2.922	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R7|dffs[14]	clock	clock	16.000	-0.095	12.998
28	2.923	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R4|dffs[14]	clock	clock	16.000	-0.095	12.997
29	2.927	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R4|dffs[9]	clock	clock	16.000	-0.095	12.993
30	2.931	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R1|dffs[9]	clock	clock	16.000	-0.087	12.997
31	2.939	lpm_ff:write_next_reg|dffs[0]	registerfile:regfile|lpm_ff:R1|dffs[3]	clock	clock	16.000	-0.088	12.988
32	2.943	lpm_ff:write_next_reg|dffs[0]	registerfile:regfile|lpm_ff:R6|dffs[3]	clock	clock	16.000	-0.095	12.977
33	2.946	lpm_ff:write_next_reg|dffs[0]	registerfile:regfile|lpm_ff:R5|dffs[3]	clock	clock	16.000	-0.095	12.974
34	2.960	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R6|dffs[4]	clock	clock	16.000	-0.095	12.960
35	3.015	lpm_ff:write_next_reg|dffs[0]	registerfile:regfile|lpm_ff:R7|dffs[3]	clock	clock	16.000	-0.088	12.912
36	3.022	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R4|dffs[11]	clock	clock	16.000	-0.095	12.898
37	3.023	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R7|dffs[11]	clock	clock	16.000	-0.095	12.897
38	3.028	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R4|dffs[4]	clock	clock	16.000	-0.095	12.892
39	3.028	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R7|dffs[4]	clock	clock	16.000	-0.088	12.899
40	3.029	lpm_ff:write_next_reg|dffs[3]	registerfile:regfile|lpm_ff:R5|dffs[4]	clock	clock	16.000	-0.095	12.891

The write_next register seemed to cause significant delay to get to the register file, due to the large amount of combinational logic due to logic gates and MUXes. This was a bottleneck to the maximum clock frequency. Pipeline stages could have been used to store the value at intermediate stages in a register and change the state machine so that each stage or cycle only involves little propagation delay. This would increase the maximum clock frequency, but also increase the setup time is takes to fully take advantage of this pipeline. These pipeline stage registers could be added at the output of the data RAM as this seemed to also introduce significant slack, as well as between the register file and ALU.

9 Conclusion/Extension

Making the CPU was very interesting as it built off the work done during term 2 very well. One could see the CPU becoming increasingly advanced as it incorporated instructions from modern ISAs and computer architectures. The CPU met the benchmark tests that were written in C++, so the link between programming and computer architecture could be explored. Meeting the design goal of creating a general CPU was satisfying as there are many different applications that can be tested.

There are several ways for the CPU to improve and to investigate further if there was more time. Firstly, in terms of the memory blocks, a dual port RAM could be tested, as this would

probably be very similar to the current setup, while also being more compact. To make the CPU more general, a signed multiplier could be implemented with slightly more hardware to multiply signed numbers. More complex implementations of a multiplier could also be investigated to attempt to decrease the number of cycles it takes. Different implementations of the Fibonacci benchmark could also be tested, as the results showed that recursion is very inefficient execution time scales exponentially with the input parameter. An assembler for the invented ISA could also be made to make it easier to convert from assembly to machine code.

In terms of optimisation, pipeline stages would have been useful to limit the effect of propagation delay on the max clock frequency. It would be interesting to investigate how much the maximum clock frequency would improve due to the added intermediate stages lowering the propagation delay and how this would improve the execution times of the benchmarks.

10 Link to Github

https://github.com/alexpondaven/CPU

Send login to ap2619@ic.ac.uk for permission to Github.

11 References

[1] Scott Thornton,"What's the difference between Von-Neumann and Harvard architectures" March 8th 2018 [online]MICROCONTROLLERTIPS available at: <https://www.microcontrollertips.com/difference-between-von-neumann-and-harvard-architectures/>

[3] Clarke, T. Class Lecture, Topic "LEC9.pdf", Department Of Electrical & Electronic Engineering, Imperial College London, UK,2020. Available at: <https://intranet.ee.ic.ac.uk/t.clarke/arch/html16/lect16/lec9.pdf>

[4] ARM QRC 0006E, Thumb 16-bit Instruction Set Qucik Reference Card [online] available at: <http://infocenter.arm.com/help/topic/com.arm.doc.qrc0006e/QRC0006_UAL16.pdf>

[5] Deepak Bordiya, and Lalit Bandil, "*Comparative Analysis Of Multipliers*" International Journal of Engineering Research & Technology. Volume 2 Issue 9, September - 2013, pp. 1437– 1441. [Accessed 8 June 2020]. [Online]. Available at: <https://www.ijert.org/research/comparative-analysis-of-multipliers-serial-and-parallel-with-radix-based-on-booth-algoritham-IJERTV2IS90625.pdf>

[6]Gim P. Hom, Joe Steinmyer, Class Lecture, Topic: "*Arithmetic Circuits & Multipliers*" 6.111 Introductory Digital Systems Laboratory, Massachusetts Institute of Technology, MA, Fall, 2017.Avaiable at: <http://web.mit.edu/6.111/www/f2017/handouts/L08.pdf>

[7] Wikipedia, Linear Congruential generator.[online] available at: <https://en.wikipedia.org/wiki/Linear_congruential_generator>

[8]Linear Congruenrial Generator I section two,Cornell Department of Mathematics. [online] available at: <http://pi.math.cornell.edu/~mec/Winter2009/Luo/Linear%20Congruential%20Generator/linear%20congruential%20gen1.html>

[9]Torsten Grust, Class Lecture, Topic "*Pipelining*" Database Systems and Modern CPU Architecture, Eberhard Karls University of Tübingen, German, 2009. Available at: <https://db.inf.uni-tuebingen.de/staticfiles/teaching/ss09/dbcpu/dbms-cpu-2.pdf>

[10] ATMEL, ww1.microchip.com. 2020. [online] Available at: <http://ww1.microchip.com/downloads/en/devicedoc/atmel-0856-avr-instruction-set-manual.pdf> [Accessed 1 June 2020].

[11] S3-eu-west-1.amazonaws.com. 2020. *MIPS32® Instruction Set Quick Reference*. [online] Available at: <https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00565-2B-MIPS32-QRC-01.01.pdf> [Accessed 28 May 2020].

Glossary, Grefer. 2020. *Risc (Reduced Instruction Set Computer)*. [online] Gartner. Available at: <https://www.gartner.com/en/information-technology/glossary/risc-reduced-instruction-set-computer> [Accessed 19 May 2020].

[13] Clarke, T. Class Lecture, Topic "LEC10.pdf", Department Of Electrical & Electronic Engineering, Imperial College London, UK,2020. Available at: <https://intranet.ee.ic.ac.uk/t.clarke/arch/html16/lect16/lec10.pdf>

12 Appendix

Appendix 1: Unpipelined ALU1

```
module alu1
(
    input [15:0] rddata, // destination register Rd data
    input [15:0] rsdata, // source register Rs data
    input [15:0] instr, // instruction word from dout
    input exec1, // From SM - for timing
    input carrystatus, // carry from carry flipflop
    input [3:0] write_next_status, // write_next output from register
    input [11:0] pc_out, // value in PC - used for BL - storing PC+1 in link register R6
    input mul_finish,
    input [15:0] LS_prod,
    input [15:0] MS_prod,

    output [15:0] dmemin,
    output [15:0] aluout, // output of ALU - written to Rd
    output carryout,
    output [3:0] write_next_out, // write_next_out[3] determines if address is written to register
    output carryen, // enables writing to carry register
    output write_next_en, // enables writing to write_next register
    output wenout, // enables writing Rd
    output [11:0] daddr, // address for data RAM (either Rs +- offset for load/store or R7 for stack instructions)
    output din_sel, // switches regfile din between aluout and ldi/ldr input
    output write_next_stp, // when a write instruction is after LDR, the PC needs to wait so that LDR finishes writing to registers
    output ldi_sel,
    output bl_sel,
    output pop_sel,
    output mul_start,

    output pc_sload,
    output pc_cnt_en,
    output ram_wren

);
//Intermediate values - need to determine instruction format
wire [3:0] op = instr[15:12];
wire sign = instr[11];
wire [4:0] offset = instr[10:6];
wire [1:0] field = instr[11:10]; // Used to determine shift type, JMP type and for cin field
wire [3:0] cond = instr[9:6]; // Determines number of shifts (can be used for other instructions)
wire cwen = instr[9];


// Opcodes
wire ldi = !op [ 3 ] & !op [ 2 ] & !op [ 1 ] & !op [ 0 ];
wire ldr = !op [ 3 ] & !op [ 2 ] & !op [ 1 ] & op [ 0 ];
wire str = !op [ 3 ] & !op [ 2 ] & op [ 1 ] & !op [ 0 ];
wire mov = !op [ 3 ] & !op [ 2 ] & op [ 1 ] & op [ 0 ];
wire jmp = !op [ 3 ] & op [ 2 ] & !op [ 1 ] & !op [ 0 ];
wire add = !op [ 3 ] & op [ 2 ] & !op [ 1 ] & op [ 0 ];
wire sub = !op [ 3 ] & op [ 2 ] & op [ 1 ] & !op [ 0 ];
wire mul = !op [ 3 ] & op [ 2 ] & op [ 1 ] & op [ 0 ];
wire bl = op [ 3 ] & !op [ 2 ] & !op [ 1 ] & !op [ 0 ];
wire ldmfd = op [ 3 ] & !op [ 2 ] & !op [ 1 ] & op [ 0 ];
wire stmfd = op [ 3 ] & !op [ 2 ] & op [ 1 ] & !op [ 0 ];
wire stp = op [ 3 ] & !op [ 2 ] & op [ 1 ] & op [ 0 ];
```

```

// Status FF bits:

// write_next tells next instruction that the data that is now at dout can be written into the Rs of the previous instruction (for load instructions)
// if write_next_flag is already 1, set to 0 (if it is not another ldr), otherwise set to 1 if it is an ldr or ldmfd instruction
assign write_next_flag = (write_next_status[3] & (~ldr | ldmfd)) ? 0 : (ldr | ldmfd);


// carryen enables writing to carry register - writes when cwen is enabled for add, sub, and mov
assign carryen = exec1 & cwen & (add | sub | mov);

// write_next_en enables writing to write_next register - needs to update for every instruction during exec1 so that it returns to 0
assign write_next_en = exec1;

// carryout equal to alucout if not a shift - note the special case of rsdata[0] for LSR or ASR (MOV instruction)
assign carryout = (mov & ((~field[1] & field[0]) | (field[1] & ~field[0]))) ? rsdata[0] : alucout;

// Output to write_next register
assign write_next_out = {write_next_flag, instr[5:3]};

// write_next_stp stop PC from counting up so instruction can finish before next instruction
assign write_next_stp = write_next_status[3] & ~(ldr | str | jmp | ldmfd | stp) | write_next_status[3] & ldmfd;




// Definition of wires and regs
wire cin; // carry input from instruction
wire shiftin; // shift in to MSB for XSR
wire alucout; // carry from alusum calculation

reg [16:0] alusum; // ALU sum needs extra bit for carry


// ALU output Calculations
assign alucout = alusum[16]; // carry bit
assign daddr = ldmfd ? (alusum[11:0]-1) : alusum[11:0]; // offset address used for addressing data memory or stores R7 value for stack manipulation
assign aluout = alusum[15:0]; // 16 bit sum

assign dmemin = stmfd ? rsdata : rddata; // input to data memory is q2 for stmfd, q1 for str


// wenout active for ldi, mov, add, sub, bl, ldmfd, stmfd (ldr not written during exec1, written in next cycle)
assign wenout = exec1 & (ldi | mov | add | sub | bl | ldmfd | stmfd | stp & write_next_status[3]);


//cin determined by cin or "type" field in instruction word
assign cin = (~field[1] & field[0]) | (field[1] & ~field[0] & carrystatus) | (field[1] & field[0] & rsdata[15]);

```

```
105
106   // din_sel should be on if writing back to registers from alu (mov, add, sub, bl, ldmfd, stmfd)
107   // If ldr is writing, din should not be changed yet
108   assign din_sel = (mov | add | sub | bl | ldmfd | stmfd) & ~write_next_stp;
109
110   // Tells when ldi is on, want it to turn off is ldr is still writing from previous instruction
111   assign ldi_sel = ldi & ~write_next_stp;
112
113   //bl_sel tells link_MUX that it is a BL instruction and port1addr needs to be 6 to load PC+1
114   assign bl_sel = bl & ~write_next_stp;
115
116   // pop_sel turns on for POP instruction when Rs needs to be written - Rs address at port1addr
117   assign pop_sel = ldmfd & write_next_flag;
118
119   // enables mul_start for the mult_instruction
120   assign mul_start = mul; // can be started even though write_next_stp is enabled (ldr is writing in this cycle)
121
122
123
124   // conditional operators - compares Rd and comparator (in cond) - used for if jump should occur
125   wire eq = (rddata == cond); // Rd == comparator
126   wire mi = (rddata < cond); // Rd < comparator
127
128   // change jmp cond to determine if jmp will occur depending on comparison and jump condition in instruction word
129   wire jmp_cond = jmp & ((~field[1] & ~field[0]) | (~field[1] & field[0] & eq) | (field[1] & ~field[0] & mi) | (field[1] & field[0] & ~eq & ~mi));
130
131   // PC and RAM control signals
132   assign pc_sload = exec1 & (jmp_cond | bl);
133   assign pc_cnt_en = exec1 & ~(jmp_cond | stp | bl | write_next_stp);
134   assign ram_wren = exec1 & (str | stmfd);
135
136
137   // determine alusum - need to change opcodes
138   always @(*) begin
139         case (op)
140         4'b0001, 4'b0010 : alusum = sign ? (rsdata - {11'b0,offset}) : (rsdata + {11'b0,offset}); // LDR and STR: calculate daddr
141
142         4'b0011 : begin
143                         case (instr[7:6])
144                               2'b00 : alusum = {1'b0,rsdata} + cin; // MOV
145                               2'b01 : alusum = {rsdata,cin}; // LSL
146                               2'b10 : alusum = {rsdata[0], cin, rsdata[15:1]};// Right shift - cin determines LSR or ASR
147                               2'b11 : alusum = instr[0] ? ({1'b0,MS_prod}+{1'b0,rddata}+cin) : ({1'b0,LS_prod}+{1'b0,rddata}+cin);// Multiply product (leas
148                               default : alusum = {1'b0,rsdata} + cin; // MOV
149                   endcase;
150               end
151         4'b0101 : alusum = {1'b0,rddata} + {1'b0,rsdata} + cin;// ADD
152         4'b0110 : alusum = {1'b0,rddata} + {1'b0,~rsdata} + cin;// SUB
153
154         4'b1000 : alusum = {5'b00000,pc_out} + 1; // BL
155
156         4'b1001 : alusum = {1'b0,rddata} + 1; // POP (LDMFD)
157         4'b1010 : alusum = {1'b0,rddata} - 1; // PUSH (STMFD)
158
159         default : alusum = 0;
160     endcase;
```

Appendix 2: Pipelined ALU1 differences

```
93    // ALU output calculations
94    assign alucout = alusum[16]; // carry bit
95    assign daddr = ldmfd ? (alusum[11:0]-1) : alusum[11:0]; // offset address used for addressing data memory or stores R7 value for stack manipulation
96    assign aluout = alusum[15:0]; // 16 bit sum
97
98    assign dmemin = stmfd ? rsdata : rddata; // input to data memory is q2 for stmfd, q1 for str
99
100
101   // wenout active for ldi, mov, add, sub, bl, ldmfd, stmfd (ldr not written during exec1, written in next cycle)
102   assign wenout = exec1 & ~write_next_stp & (ldi | mov | add | sub | bl | ldmfd | stmfd | stp & write_next_status[3]);
103
104
105   //cin determined by cin or "type" field in instruction word
106   assign cin = (~field[1] & field[0]) | (field[1] & ~field[0] & carrystatus) | (field[1] & field[0] & rsdata[15]);
107
108   // din_sel should be on if writing back to registers from alu (mov, add, sub, bl, ldmfd, stmfd)
109   // If ldr is writing, din should not be changed yet
110   assign din_sel = (mov | add | sub | bl | ldmfd | stmfd) & ~write_next_stp;
111
112   // Tells when ldi is on, want it to turn off is ldr is still writing from previous instruction
113   assign ldi_sel = ldi & ~write_next_stp;
114
115   //bl_sel tells link_MUX that it is a BL instruction and port1addr needs to be 6 to load PC+1
116   assign bl_sel = bl & ~write_next_stp;
117
118   // pop_sel turns on for POP instruction when Rs needs to be written - Rs address at port1addr
119   assign pop_sel = ldmfd & write_next_flag;
120
121   // enables mul_start for the mult_instruction
122   assign mul_start = mul & ~write_next_stp; // can be started even though write_next_stp is enabled (ldr is writing in this cycle)
123
124
125
126   // conditional operators - compares Rd and comparator (in cond) - used for if jump should occur
127   wire eq = (rddata == cond); // Rd == comparator
128   wire mi = (rddata < cond); // Rd < comparator
129
130   // change jmp cond to determine if jmp will occur depending on comparison and jump condition in instruction word
131   wire jmp_cond = jmp & ((~field[1] & ~field[0]) | (~field[1] & field[0] & eq) | (field[1] & ~field[0] & mi) | (field[1] & field[0] & ~eq & ~mi));
132
133   // PC and RAM control signals
134   assign pc_sload = exec1 & (jmp_cond | bl);
135   assign pc_cnt_en = exec1 & ~(jmp_cond | stp | bl | write_next_stp);
136   assign ram_wren = exec1 & (str | stmfd);
137   assign add_sel = exec1 & ~(stp | jmp_cond | bl | write_next_stp);
138
139
140   // determine alusum - need to change opcodes
141   always @(*) begin
142         case (op)
143         4'b0001, 4'b0010 : alusum = sign ? (rsdata - {11'b0,offset}) : (rsdata + {11'b0,offset}); // LDR and STR: calculate daddr
144
145         4'b0011 : begin
146                         case (instr[7:6])
147
```

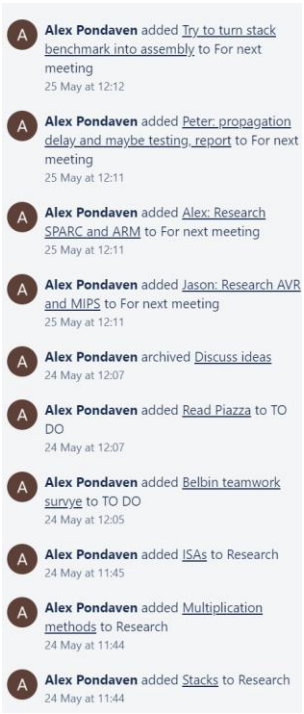
Appendix 3: Meeting notes

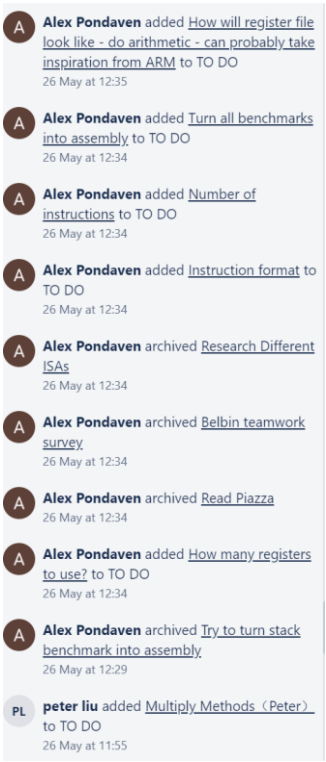
Date	Discussion	Alex	Peter	Jason
25/5	• Discussed stack and shared research on AVR and SPARC • Went through fibonacci benchmark			

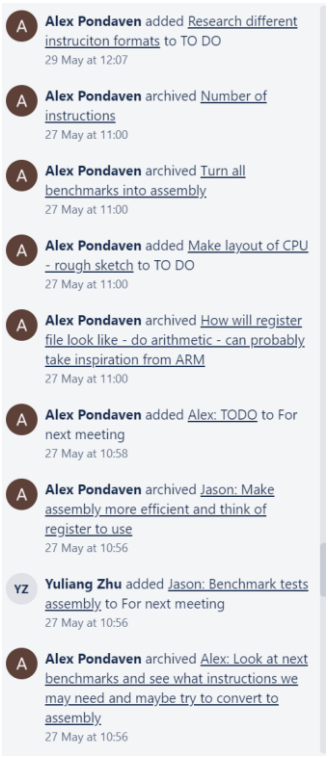
26/6	• Discussed Jason's assembly code for fib benchmark • Went over stacks • Discussed number of registers and instruction format ○ Decided it is best to decide this once all benchmarks have been turned into assembly and we know what instructions will be needed	Look at next benchmarks and see what instructions we may need and convert to assembly	Research multiplication methods	Make assembly more efficient and think of register to use
27/5	• Went over assembly code for LCG and linked list benchmarks • Discussed multiplication methods • Made a list of instructions using benchmarks	Think of implementation of instructions (DECA oral as well)	Continue multiplication method research	Reduce number of instructions used in fib
29/5	• Went through Jason's in detail implementation of stack instructions - may need new instructions • Made plan of CPU - went through each instruction to determine how it may be implemented - not sure about branch (BL) and LDMB, STMB instructions for fib	Branch and link hardware implementation	Research multiply methods	LDMB (POP) implementation
31/5	• Discussed LDMFD instruction for loading and increasing stack pointer - this could be done with a bit in instruction word specifying if stack pointer should be changed • Discussed instruction word format • Got stuck on trying to make LDR one cycle - not sure if this is possible	Look at different instruction formats (e.g. Thumb) and determine instruction formats	Force block and multiplication in ALU	LDR implementation in different ISAs
5/6	• Discussed multiplication - number of cycles ○ Instruction or subroutine?	Testing LDR	Parallel multiplier research	ISA table and Branch logic

6/6	• Discussed parallel multiplier ideas
	  ○ Shift/add
	  ○ Parallel - only helps with multiple consecutive multiplies
	• How will we store the 2 registers afterwards
	  ○ 2 registers only for multiply result			

7/6	• Github merging problems • Peter made progress on multiply report • Discussed how we could split up testing work later after github is fixed	Fix github and finish testing all instructions	Implement synchronous multiplier	Report structure and revise hardware








Alex Pondaven added Think of how to implement LDI (12 bits in instruction word but also address the register needed) - LDI may only work for one register - extra cycles to TO DO
2 Jun at 12:42
Alex Pondaven archived Delete test files in github
1 Jun at 22:52
Alex Pondaven archived Jason: Stack Register
1 Jun at 22:52
Alex Pondaven added Delete test files in github to TO DO
1 Jun at 19:22
Yuliang Zhu added Jason: Stack Register to For next meeting
31 May at 10:24
Alex Pondaven archived Stacks/caches
30 May at 12:39
Alex Pondaven archived Make layout of CPU - rough sketch
30 May at 12:39
Alex Pondaven archived Multiply Methods，Harvard （Peter）
30 May at 12:39
Alex Pondaven archived How many registers to use?
30 May at 12:39
Alex Pondaven archived Instruction format
30 May at 12:39
Alex Pondaven archived Do we need MUX1?
3 Jun at 11:34
Alex Pondaven archived Make a decoder table to determine logic
3 Jun at 11:34
Alex Pondaven archived Separate stack pointer
3 Jun at 11:34
Alex Pondaven archived Think about SM
3 Jun at 11:34
Alex Pondaven archived Find decoder signals
3 Jun at 11:34
Alex Pondaven added Do we need MUX1? to TO DO
2 Jun at 13:09
Alex Pondaven added Make a decoder table to determine logic to TO DO
2 Jun at 13:04
Alex Pondaven added Find decoder signals to TO DO
2 Jun at 13:04
Alex Pondaven added Think about SM to TO DO
2 Jun at 13:03
Alex Pondaven added Separate stack pointer to TO DO
2 Jun at 13:03
Alex Pondaven moved What should the last cin field be? CMSB (as in ARMish) is not very useful now since shifts are being implemented with MOV instruction from TO DO to Questions
3 Jun at 18:03
Alex Pondaven moved Is there any way to do LDI for all registers? from TO DO to Questions
3 Jun at 18:03
Alex Pondaven added Questions to this board
3 Jun at 18:03
Alex Pondaven added Do we need to reference ARMish design as we took inspiration from there? to TO DO
3 Jun at 18:02
Alex Pondaven added We basically have 2 decoders now in ALU and decoder outside of this block. Could SM be put inside of regfile_alu and then make all control signals come from this block? Would be more clear to TO DO
3 Jun at 17:07
Alex Pondaven added What should the last cin field be? CMSB (as in ARMish) is not very useful now since shifts are being implemented with MOV instruction to TO DO
3 Jun at 16:17
Alex Pondaven added Is there any way to do LDI for all registers? to TO DO
3 Jun at 15:57
Alex Pondaven added Remov eq and mi from status register - not needed to TO DO
5 Jun at 15:31
Alex Pondaven added Make table to summarise instruction formats to TO DO
4 Jun at 17:32
Alex Pondaven added Update input to PC from q2 to TO DO
4 Jun at 10:51
Alex Pondaven added Status register needs to store the comparison between Rd and Rs for JMP instructions to TO DO
4 Jun at 10:31
Alex Pondaven added Read through stack lecture to TO DO
3 Jun at 19:14
Alex Pondaven archived ALU
3 Jun at 18:04
Alex Pondaven added ALU to TO DO
3 Jun at 18:04
Alex Pondaven moved Do we need to reference ARMish design as we took inspiration from there? from TO DO to Questions
3 Jun at 18:04
Alex Pondaven moved We basically have 2 decoders now in ALU and decoder outside of this block. Could SM be put inside of regfile_alu and then make all control signals come from this block? Would be more clear from TO DO to Questions
3 Jun at 18:04
Alex Pondaven added Change instr. mem to RAM to TO DO
9 Jun at 10:01
Alex Pondaven archived Update input to PC from q2
8 Jun at 15:42
Alex Pondaven added Forgot a bit in MOV code - need to update this to TO DO
8 Jun at 11:13
Alex Pondaven added Write tests for all instructions to TO DO
8 Jun at 11:03
Alex Pondaven archived Do we need to reference ARMish design as we took inspiration from there?
7 Jun at 17:01
Alex Pondaven archived Is there any way to do LDI for all registers?
7 Jun at 17:01
Alex Pondaven archived Make table to summarise instruction formats (include status register)
7 Jun at 17:01
Alex Pondaven archived Remov eq and mi from status register - not needed
7 Jun at 17:01
Alex Pondaven archived Status register needs to store the comparison between Rd and Rs for JMP instructions
6 Jun at 10:32
Alex Pondaven added Implement multiplication block , adding block, adding output register, to TO DO
10 Jun at 11:55
Alex Pondaven added Test fibonnacci to TO DO
10 Jun at 11:27
Alex Pondaven added Add LDMFD and STMFD to TO DO
10 Jun at 11:27
Alex Pondaven archived We basically have 2 decoders now in ALU and decoder outside of this block. Could SM be put inside of regfile_alu and then make all control signals come from this block? Would be more clear
9 Jun at 20:45
Alex Pondaven archived What should the last cin field be? CMSB (as in ARMish) is not very useful now since shifts are being implemented with MOV instruction
9 Jun at 20:45
Alex Pondaven archived Forgot a bit in MOV code - need to update this
9 Jun at 20:44
Alex Pondaven archived Write tests for all instructions
9 Jun at 20:44
Alex Pondaven added Change STP opcode to A rather than B, as stack was combined, so A is empty now to TO DO
9 Jun at 20:44

Figure 1

Appendix 4

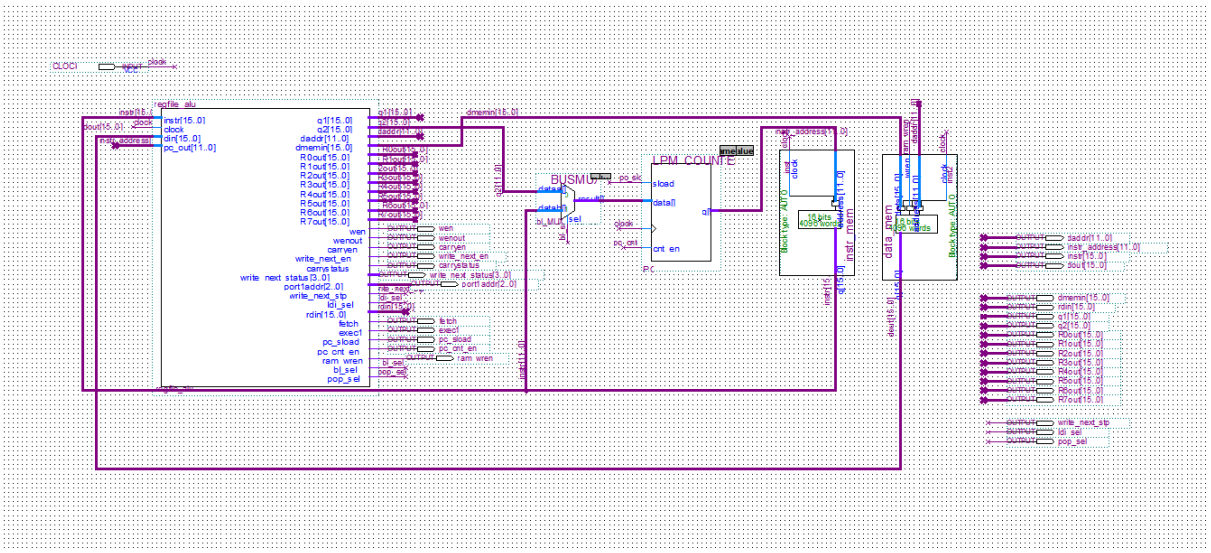


Figure 2

Appendix 5

```
module multwithRadix4(prod,finish,A,B,start,clk);

   input [15:0]  A, B;
   input         start, clk;
   output        prod;
   output        finish;

   reg [4:0]     count;
   wire          finish = !count;


   reg [32:0]    product;
   wire [31:0]   prod = product[31:0];


   reg [17:0]    tmp;

   initial count = 0;

   wire [17:0]   A_X_1 = {1'b0,A};
   wire [17:0]   A_X_2 = {A,1'b0};
   wire [17:0]   A_X_3 = A_X_2 + A_X_1;

   always @( posedge clk )

     if( finish && start ) begin

        count     = 8;
        product = { 16'd0, B };

     end else if( count ) begin

        case ( {product[1:0]} )
          2'b00: tmp = {2'b0, product[31:16] };
          2'b01: tmp = {2'b0, product[31:16] } + A_X_1;
          2'b10: tmp = {2'b0, product[31:16] } + A_X_2;
          2'b11: tmp = {2'b0, product[31:16] } + A_X_3;
        endcase

        product = { tmp, product[15:2] };
        count     = count - 1;

     end

endmodule
```

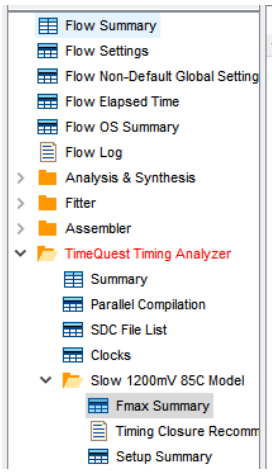
Figure 10

Appendix 6

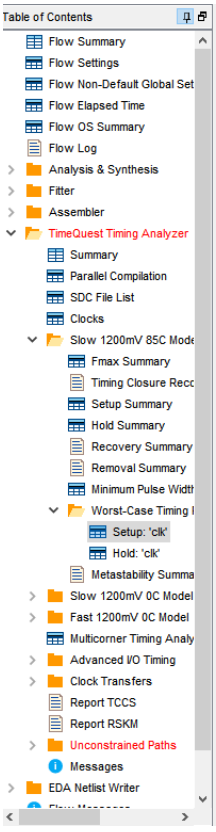
```
module multwithRadix8(prod,finish,A,B,start,clk);

   input [15:0]  A, B;
   input         start, clk;
   output        prod;
   output        finish;

   reg [2:0]     count;
   wire          finish = !count;


   reg [32:0]    product;
   wire [31:0]   prod = product[31:0];


   reg [19:0]    tmp;

   initial count = 0;

   wire [19:0]   A_X_1 = {1'b0,A};
   wire [19:0]   A_X_2 = {A,1'b0};
   wire [19:0]   A_X_3 = A_X_2 + A_X_1;
   wire [19:0]   A_X_4 = {A,2'b0};
   wire [19:0]   A_X_5 = A_X_4 + A_X_1;
   wire [19:0]   A_X_6 = A_X_4 + A_X_2;
   wire [19:0]   A_X_7 = A_X_4 + A_X_3;
   wire [19:0]   A_X_8 = {A,3'b0};
   wire [19:0]   A_X_9 = A_X_8 + A_X_1;
   wire [19:0]   A_X_10 = A_X_8 + A_X_2;
   wire [19:0]   A_X_11 = A_X_8 + A_X_3;
   wire [19:0]   A_X_12 = A_X_8 + A_X_4;
   wire [19:0]   A_X_13 = A_X_8 + A_X_5;
   wire [19:0]   A_X_14 = A_X_8 + A_X_6;
   wire [19:0]   A_X_15 = A_X_8 + A_X_7;

   always @( posedge clk )

     if( finish && start ) begin

        count     = 4:

        product = { 16'd0, B };

     end else if( count ) begin

        case ( {product[3:0]} )

           4'b0000: tmp = {4'b0, product[31:16] };
           4'b0001: tmp = {4'b0, product[31:16] } + A_X_1;
           4'b0010: tmp = {4'b0, product[31:16] } + A_X_2;
           4'b0011: tmp = {4'b0, product[31:16] } + A_X_3;
           4'b0100: tmp = {4'b0, product[31:16] } + A_X_4;
           4'b0101: tmp = {4'b0, product[31:16] } + A_X_5;
           4'b0110: tmp = {4'b0, product[31:16] } + A_X_6;
           4'b0111: tmp = {4'b0, product[31:16] } + A_X_7;
           4'b1000: tmp = {4'b0, product[31:16] } + A_X_8;
           4'b1001: tmp = {4'b0, product[31:16] } + A_X_9;
           4'b1010: tmp = {4'b0, product[31:16] } + A_X_10;
           4'b1011: tmp = {4'b0, product[31:16] } + A_X_11;
           4'b1100: tmp = {4'b0, product[31:16] } + A_X_12;
           4'b1101: tmp = {4'b0, product[31:16] } + A_X_13;
           4'b1110: tmp = {4'b0, product[31:16] } + A_X_14;
           4'b1111: tmp = {4'b0, product[31:16] } + A_X_15;
        endcase

        product = { tmp, product[15:4] };
        count     = count - 1;

     end

endmodule
```

Figure 11

Appendix 7

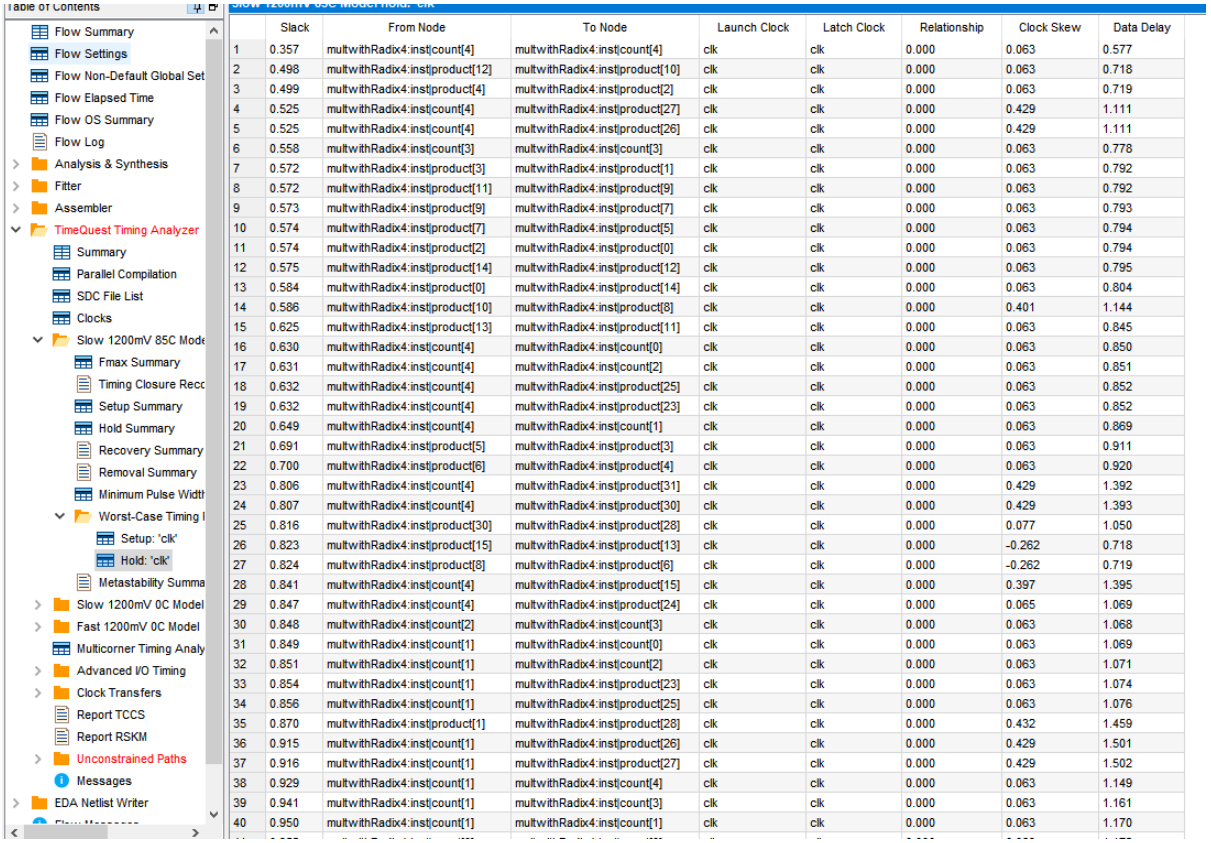


	Fmax	Restricted Fmax	Clock Name	Note
1	277.39 MHz	250.0 MHz	clk	limit due to minimum period restriction (max I/O toggle rate)



Slow 1200mV 85C Model Setup: 'clk'

	Slack	From Node	To Node	Launch Clock	Latch Clock	Relationship	Clock Skew	Data Delay
1	0.395	multwithRadix4:inst|product[17]	multwithRadix4:inst|product[22]	clk	clk	4.000	-0.064	3.556
2	0.425	multwithRadix4:inst|product[18]	multwithRadix4:inst|product[22]	clk	clk	4.000	-0.064	3.526
3	0.448	multwithRadix4:inst|product[23]	multwithRadix4:inst|product[25]	clk	clk	4.000	-0.063	3.504
4	0.456	multwithRadix4:inst|product[23]	multwithRadix4:inst|product[23]	clk	clk	4.000	-0.063	3.496
5	0.483	multwithRadix4:inst|product[17]	multwithRadix4:inst|product[18]	clk	clk	4.000	-0.064	3.468
6	0.498	multwithRadix4:inst|product[19]	multwithRadix4:inst|product[21]	clk	clk	4.000	-0.063	3.454
7	0.513	multwithRadix4:inst|product[18]	multwithRadix4:inst|product[18]	clk	clk	4.000	-0.064	3.438
8	0.514	multwithRadix4:inst|product[16]	multwithRadix4:inst|product[21]	clk	clk	4.000	-0.064	3.437
9	0.516	multwithRadix4:inst|product[17]	multwithRadix4:inst|product[21]	clk	clk	4.000	-0.064	3.435
10	0.551	multwithRadix4:inst|product[18]	multwithRadix4:inst|product[25]	clk	clk	4.000	-0.067	3.397
11	0.601	multwithRadix4:inst|product[20]	multwithRadix4:inst|product[21]	clk	clk	4.000	-0.063	3.351
12	0.612	multwithRadix4:inst|product[16]	multwithRadix4:inst|product[17]	clk	clk	4.000	-0.064	3.339
13	0.614	multwithRadix4:inst|product[17]	multwithRadix4:inst|product[17]	clk	clk	4.000	-0.064	3.337
14	0.619	multwithRadix4:inst|product[21]	multwithRadix4:inst|product[21]	clk	clk	4.000	-0.063	3.333
15	0.630	multwithRadix4:inst|product[18]	multwithRadix4:inst|product[21]	clk	clk	4.000	-0.064	3.321
16	0.635	multwithRadix4:inst|product[17]	multwithRadix4:inst|product[25]	clk	clk	4.000	-0.067	3.313
17	0.640	multwithRadix4:inst|product[22]	multwithRadix4:inst|product[22]	clk	clk	4.000	-0.064	3.311
18	0.640	multwithRadix4:inst|product[26]	multwithRadix4:inst|product[25]	clk	clk	4.000	-0.429	2.946
19	0.642	multwithRadix4:inst|product[17]	multwithRadix4:inst|product[20]	clk	clk	4.000	-0.064	3.309
20	0.647	multwithRadix4:inst|product[23]	multwithRadix4:inst|product[24]	clk	clk	4.000	-0.061	3.307
21	0.653	multwithRadix4:inst|product[17]	multwithRadix4:inst|product[26]	clk	clk	4.000	0.284	3.646
22	0.661	multwithRadix4:inst|product[19]	multwithRadix4:inst|product[25]	clk	clk	4.000	-0.066	3.288
23	0.665	multwithRadix4:inst|product[16]	multwithRadix4:inst|product[25]	clk	clk	4.000	-0.067	3.283
24	0.668	multwithRadix4:inst|product[19]	multwithRadix4:inst|product[22]	clk	clk	4.000	-0.063	3.284
25	0.669	multwithRadix4:inst|product[19]	multwithRadix4:inst|product[23]	clk	clk	4.000	-0.066	3.280
26	0.672	multwithRadix4:inst|product[18]	multwithRadix4:inst|product[20]	clk	clk	4.000	-0.064	3.279
27	0.672	multwithRadix4:inst|product[17]	multwithRadix4:inst|product[24]	clk	clk	4.000	-0.065	3.278
28	0.684	multwithRadix4:inst|product[17]	multwithRadix4:inst|product[23]	clk	clk	4.000	-0.067	3.264
29	0.685	multwithRadix4:inst|product[16]	multwithRadix4:inst|product[23]	clk	clk	4.000	-0.067	3.263
30	0.688	multwithRadix4:inst|product[17]	multwithRadix4:inst|product[30]	clk	clk	4.000	0.284	3.611
31	0.688	multwithRadix4:inst|product[23]	multwithRadix4:inst|product[22]	clk	clk	4.000	-0.060	3.267
32	0.699	multwithRadix4:inst|product[18]	multwithRadix4:inst|product[26]	clk	clk	4.000	0.284	3.600
33	0.705	multwithRadix4:inst|product[20]	multwithRadix4:inst|product[22]	clk	clk	4.000	-0.063	3.247
34	0.706	multwithRadix4:inst|product[18]	multwithRadix4:inst|product[24]	clk	clk	4.000	-0.065	3.244
35	0.718	multwithRadix4:inst|product[18]	multwithRadix4:inst|product[30]	clk	clk	4.000	0.284	3.581
36	0.721	multwithRadix4:inst|product[23]	multwithRadix4:inst|product[21]	clk	clk	4.000	-0.060	3.234
37	0.722	multwithRadix4:inst|product[18]	multwithRadix4:inst|product[27]	clk	clk	4.000	0.284	3.577
38	0.728	multwithRadix4:inst|product[18]	multwithRadix4:inst|product[17]	clk	clk	4.000	-0.064	3.223
39	0.752	multwithRadix4:inst|product[20]	multwithRadix4:inst|product[25]	clk	clk	4.000	-0.066	3.197
40	0.756	multwithRadix4:inst|product[19]	multwithRadix4:inst|product[18]	clk	clk	4.000	-0.063	3.196



	Slack	From Node	To Node	Launch Clock	Latch Clock	Relationship	Clock Skew	Data Delay
1	0.357	multwithRadix4:inst|count[4]	multwithRadix4:inst|count[4]	clk	clk	0.000	0.063	0.577
2	0.498	multwithRadix4:inst|product[12]	multwithRadix4:inst|product[10]	clk	clk	0.000	0.063	0.718
3	0.499	multwithRadix4:inst|product[4]	multwithRadix4:inst|product[2]	clk	clk	0.000	0.063	0.719
4	0.525	multwithRadix4:inst|count[4]	multwithRadix4:inst|product[27]	clk	clk	0.000	0.429	1.111
5	0.525	multwithRadix4:inst|count[4]	multwithRadix4:inst|product[26]	clk	clk	0.000	0.429	1.111
6	0.558	multwithRadix4:inst|count[3]	multwithRadix4:inst|count[3]	clk	clk	0.000	0.063	0.778
7	0.572	multwithRadix4:inst|product[3]	multwithRadix4:inst|product[1]	clk	clk	0.000	0.063	0.792
8	0.572	multwithRadix4:inst|product[11]	multwithRadix4:inst|product[9]	clk	clk	0.000	0.063	0.792
9	0.573	multwithRadix4:inst|product[9]	multwithRadix4:inst|product[7]	clk	clk	0.000	0.063	0.793
10	0.574	multwithRadix4:inst|product[7]	multwithRadix4:inst|product[5]	clk	clk	0.000	0.063	0.794
11	0.574	multwithRadix4:inst|product[2]	multwithRadix4:inst|product[0]	clk	clk	0.000	0.063	0.794
12	0.575	multwithRadix4:inst|product[14]	multwithRadix4:inst|product[12]	clk	clk	0.000	0.063	0.795
13	0.584	multwithRadix4:inst|product[0]	multwithRadix4:inst|product[14]	clk	clk	0.000	0.063	0.804
14	0.586	multwithRadix4:inst|product[10]	multwithRadix4:inst|product[8]	clk	clk	0.000	0.401	1.144
15	0.625	multwithRadix4:inst|product[13]	multwithRadix4:inst|product[11]	clk	clk	0.000	0.063	0.845
16	0.630	multwithRadix4:inst|count[4]	multwithRadix4:inst|count[0]	clk	clk	0.000	0.063	0.850
17	0.631	multwithRadix4:inst|count[4]	multwithRadix4:inst|count[2]	clk	clk	0.000	0.063	0.851
18	0.632	multwithRadix4:inst|count[4]	multwithRadix4:inst|product[25]	clk	clk	0.000	0.063	0.852
19	0.632	multwithRadix4:inst|count[4]	multwithRadix4:inst|product[23]	clk	clk	0.000	0.063	0.852
20	0.649	multwithRadix4:inst|count[4]	multwithRadix4:inst|count[1]	clk	clk	0.000	0.063	0.869
21	0.691	multwithRadix4:inst|product[5]	multwithRadix4:inst|product[3]	clk	clk	0.000	0.063	0.911
22	0.700	multwithRadix4:inst|product[6]	multwithRadix4:inst|product[4]	clk	clk	0.000	0.063	0.920
23	0.806	multwithRadix4:inst|count[4]	multwithRadix4:inst|product[31]	clk	clk	0.000	0.429	1.392
24	0.807	multwithRadix4:inst|count[4]	multwithRadix4:inst|product[30]	clk	clk	0.000	0.429	1.393
25	0.816	multwithRadix4:inst|product[30]	multwithRadix4:inst|product[28]	clk	clk	0.000	0.077	1.050
26	0.823	multwithRadix4:inst|product[15]	multwithRadix4:inst|product[13]	clk	clk	0.000	-0.262	0.718
27	0.824	multwithRadix4:inst|product[8]	multwithRadix4:inst|product[6]	clk	clk	0.000	-0.262	0.719
28	0.841	multwithRadix4:inst|count[4]	multwithRadix4:inst|product[15]	clk	clk	0.000	0.397	1.395
29	0.847	multwithRadix4:inst|count[4]	multwithRadix4:inst|product[24]	clk	clk	0.000	0.065	1.069
30	0.848	multwithRadix4:inst|count[2]	multwithRadix4:inst|count[3]	clk	clk	0.000	0.063	1.068
31	0.849	multwithRadix4:inst|count[1]	multwithRadix4:inst|count[0]	clk	clk	0.000	0.063	1.069
32	0.851	multwithRadix4:inst|count[1]	multwithRadix4:inst|count[2]	clk	clk	0.000	0.063	1.071
33	0.854	multwithRadix4:inst|count[1]	multwithRadix4:inst|product[23]	clk	clk	0.000	0.063	1.074
34	0.856	multwithRadix4:inst|count[1]	multwithRadix4:inst|product[25]	clk	clk	0.000	0.063	1.076
35	0.870	multwithRadix4:inst|product[1]	multwithRadix4:inst|product[28]	clk	clk	0.000	0.432	1.459
36	0.915	multwithRadix4:inst|count[1]	multwithRadix4:inst|product[26]	clk	clk	0.000	0.429	1.501
37	0.916	multwithRadix4:inst|count[1]	multwithRadix4:inst|product[27]	clk	clk	0.000	0.429	1.502
38	0.929	multwithRadix4:inst|count[1]	multwithRadix4:inst|count[4]	clk	clk	0.000	0.063	1.149
39	0.941	multwithRadix4:inst|count[1]	multwithRadix4:inst|count[3]	clk	clk	0.000	0.063	1.161
40	0.950	multwithRadix4:inst|count[1]	multwithRadix4:inst|count[1]	clk	clk	0.000	0.063	1.170

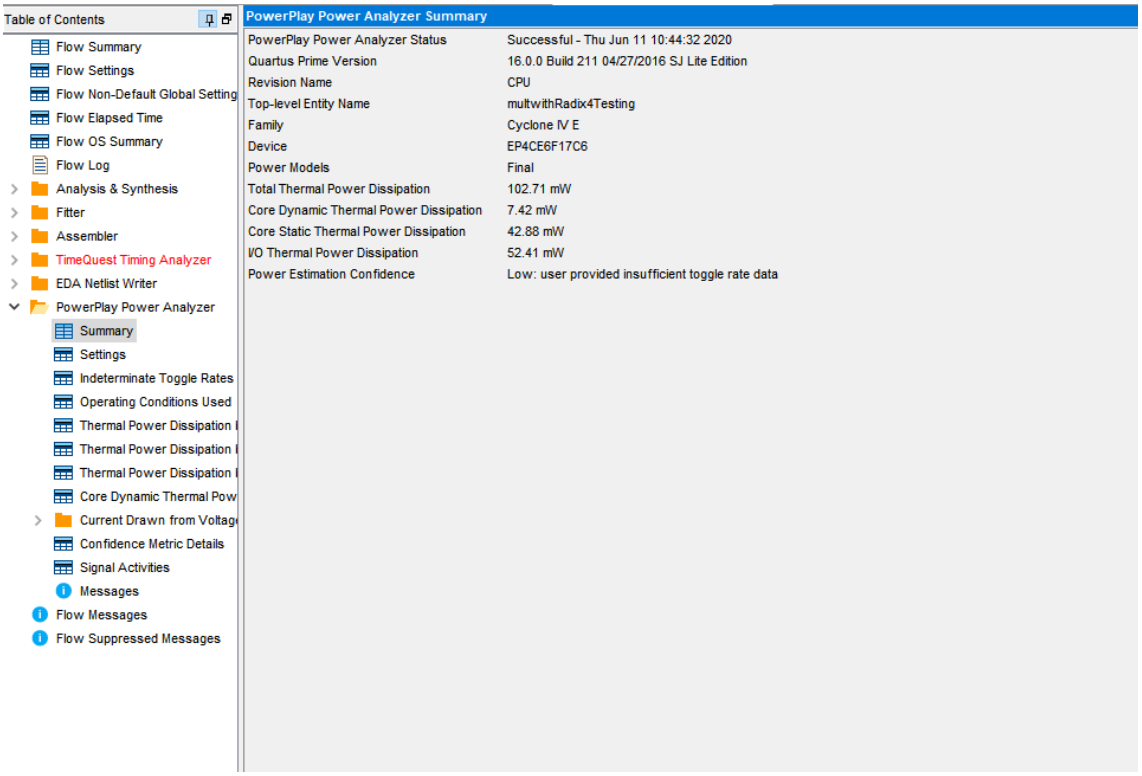
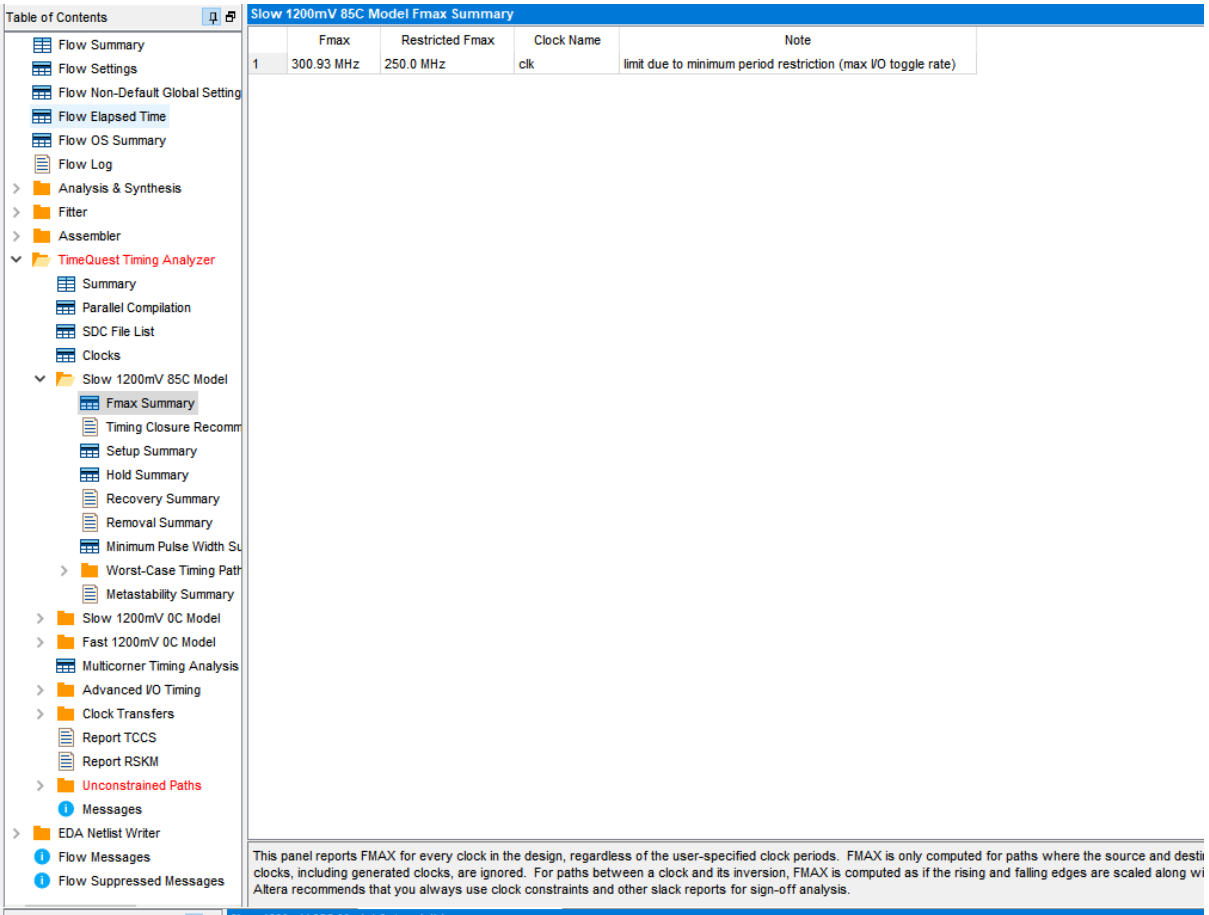
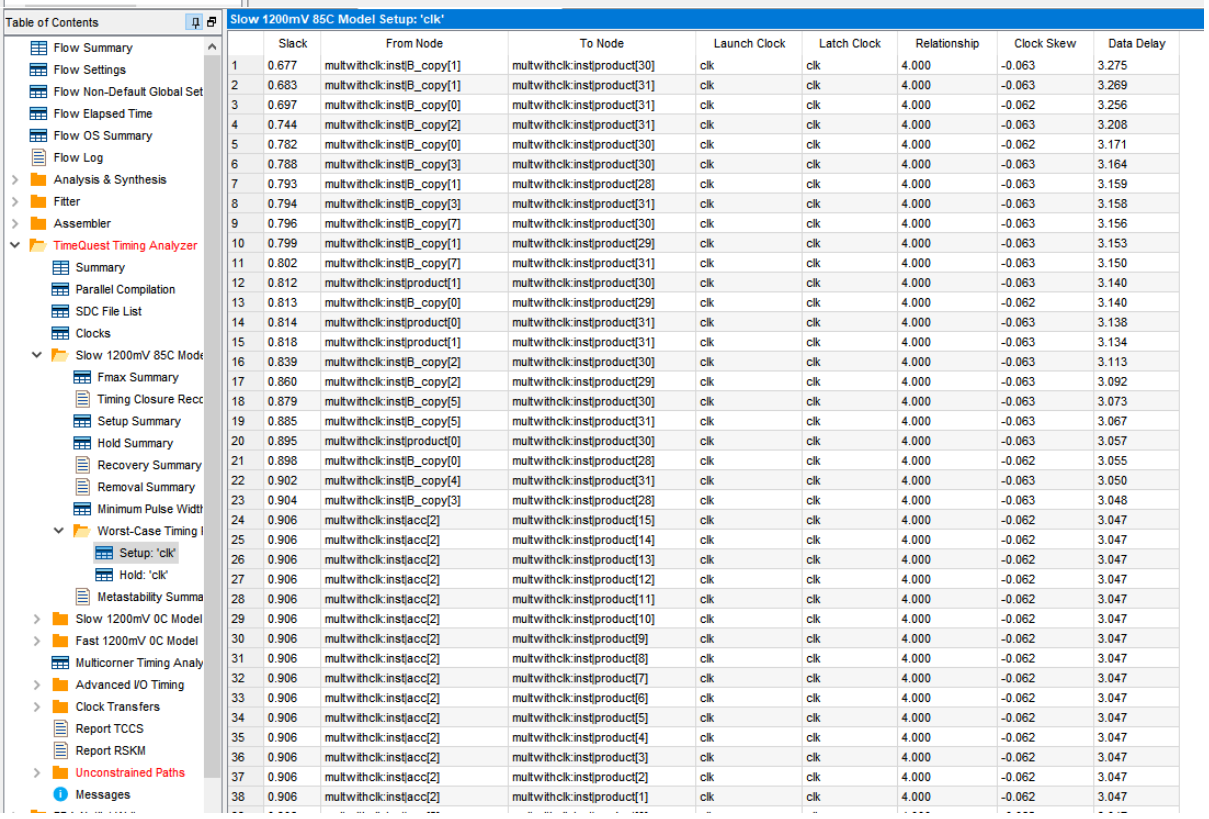


Figure analysis Radix 4

Appendix 8





	Slack	From Node	To Node	Launch Clock	Latch Clock	Relationship	Clock Skew	Data Delay
1	0.677	multwithclk:inst|B_copy[1]	multwithclk:inst|product[30]	clk	clk	4.000	-0.063	3.275
2	0.683	multwithclk:inst|B_copy[1]	multwithclk:inst|product[31]	clk	clk	4.000	-0.063	3.269
3	0.697	multwithclk:inst|B_copy[0]	multwithclk:inst|product[31]	clk	clk	4.000	-0.062	3.256
4	0.744	multwithclk:inst|B_copy[2]	multwithclk:inst|product[31]	clk	clk	4.000	-0.063	3.208
5	0.782	multwithclk:inst|B_copy[0]	multwithclk:inst|product[30]	clk	clk	4.000	-0.062	3.171
6	0.788	multwithclk:inst|B_copy[3]	multwithclk:inst|product[30]	clk	clk	4.000	-0.063	3.164
7	0.793	multwithclk:inst|B_copy[1]	multwithclk:inst|product[28]	clk	clk	4.000	-0.063	3.159
8	0.794	multwithclk:inst|B_copy[3]	multwithclk:inst|product[31]	clk	clk	4.000	-0.063	3.158
9	0.796	multwithclk:inst|B_copy[7]	multwithclk:inst|product[30]	clk	clk	4.000	-0.063	3.156
10	0.799	multwithclk:inst|B_copy[1]	multwithclk:inst|product[29]	clk	clk	4.000	-0.063	3.153
11	0.802	multwithclk:inst|B_copy[7]	multwithclk:inst|product[31]	clk	clk	4.000	-0.063	3.150
12	0.812	multwithclk:inst|product[1]	multwithclk:inst|product[30]	clk	clk	4.000	-0.063	3.140
13	0.813	multwithclk:inst|B_copy[0]	multwithclk:inst|product[29]	clk	clk	4.000	-0.062	3.140
14	0.814	multwithclk:inst|product[0]	multwithclk:inst|product[31]	clk	clk	4.000	-0.063	3.138
15	0.818	multwithclk:inst|product[1]	multwithclk:inst|product[31]	clk	clk	4.000	-0.063	3.134
16	0.839	multwithclk:inst|B_copy[2]	multwithclk:inst|product[30]	clk	clk	4.000	-0.063	3.113
17	0.860	multwithclk:inst|B_copy[2]	multwithclk:inst|product[29]	clk	clk	4.000	-0.063	3.092
18	0.879	multwithclk:inst|B_copy[5]	multwithclk:inst|product[30]	clk	clk	4.000	-0.063	3.073
19	0.885	multwithclk:inst|B_copy[5]	multwithclk:inst|product[31]	clk	clk	4.000	-0.063	3.067
20	0.895	multwithclk:inst|product[0]	multwithclk:inst|product[30]	clk	clk	4.000	-0.063	3.057
21	0.898	multwithclk:inst|B_copy[0]	multwithclk:inst|product[28]	clk	clk	4.000	-0.062	3.055
22	0.902	multwithclk:inst|B_copy[4]	multwithclk:inst|product[31]	clk	clk	4.000	-0.063	3.050
23	0.904	multwithclk:inst|B_copy[3]	multwithclk:inst|product[28]	clk	clk	4.000	-0.063	3.048
24	0.906	multwithclk:inst|acc[2]	multwithclk:inst|product[15]	clk	clk	4.000	-0.062	3.047
25	0.906	multwithclk:inst|acc[2]	multwithclk:inst|product[14]	clk	clk	4.000	-0.062	3.047
26	0.906	multwithclk:inst|acc[2]	multwithclk:inst|product[13]	clk	clk	4.000	-0.062	3.047
27	0.906	multwithclk:inst|acc[2]	multwithclk:inst|product[12]	clk	clk	4.000	-0.062	3.047
28	0.906	multwithclk:inst|acc[2]	multwithclk:inst|product[11]	clk	clk	4.000	-0.062	3.047
29	0.906	multwithclk:inst|acc[2]	multwithclk:inst|product[10]	clk	clk	4.000	-0.062	3.047
30	0.906	multwithclk:inst|acc[2]	multwithclk:inst|product[9]	clk	clk	4.000	-0.062	3.047
31	0.906	multwithclk:inst|acc[2]	multwithclk:inst|product[8]	clk	clk	4.000	-0.062	3.047
32	0.906	multwithclk:inst|acc[2]	multwithclk:inst|product[7]	clk	clk	4.000	-0.062	3.047
33	0.906	multwithclk:inst|acc[2]	multwithclk:inst|product[6]	clk	clk	4.000	-0.062	3.047
34	0.906	multwithclk:inst|acc[2]	multwithclk:inst|product[5]	clk	clk	4.000	-0.062	3.047
35	0.906	multwithclk:inst|acc[2]	multwithclk:inst|product[4]	clk	clk	4.000	-0.062	3.047
36	0.906	multwithclk:inst|acc[2]	multwithclk:inst|product[3]	clk	clk	4.000	-0.062	3.047
37	0.906	multwithclk:inst|acc[2]	multwithclk:inst|product[2]	clk	clk	4.000	-0.062	3.047
38	0.906	multwithclk:inst|acc[2]	multwithclk:inst|product[1]	clk	clk	4.000	-0.062	3.047

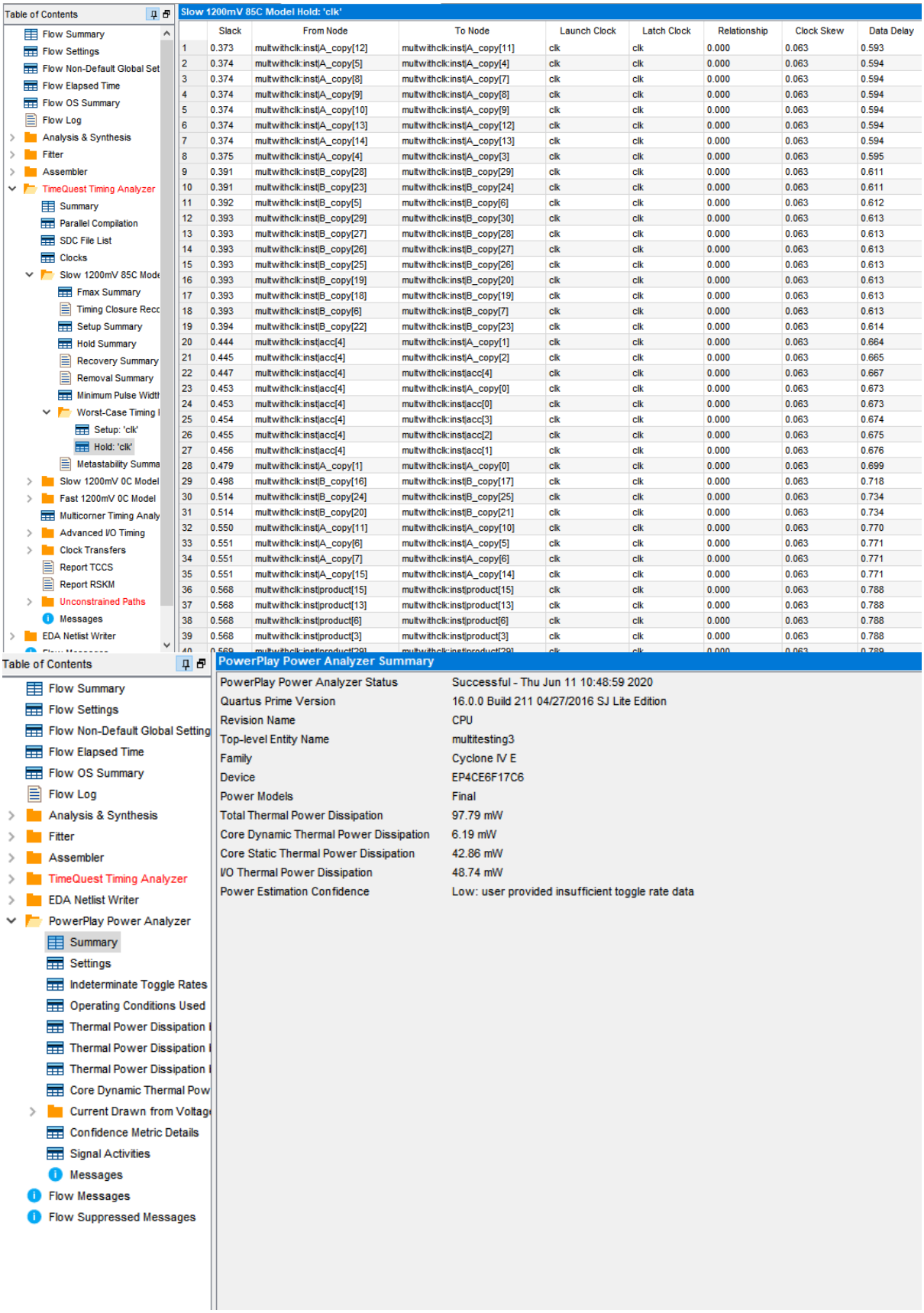


Figure analysis Radix 2

Appendix 9

The ROM implementation had a slightly higher Fmax than using RAM when testing the unpipelined versions as seen below.

Instruction ROM implementation

	Fmax	Restricted Fmax	Clock Name	Note
1	69.42 MHz	69.42 MHz	clk	

PowerPlay Power Analyzer Summary	
PowerPlay Power Analyzer Status	Successful - Thu Jun 11 17:35:15 2020
Quartus Prime Version	16.0.0 Build 211 04/27/2016 SJ Lite Edition
Revision Name	CPU
Top-level Entity Name	CPU_block_timing
Family	Cyclone IV E
Device	EP4CE6E22C6
Power Models	Final
Total Thermal Power Dissipation	129.61 mW
Core Dynamic Thermal Power Dissipation	60.57 mW
Core Static Thermal Power Dissipation	43.01 mW
I/O Thermal Power Dissipation	26.03 mW
Power Estimation Confidence	Low: user provided insufficient toggle rate data

Instruction RAM implementation:

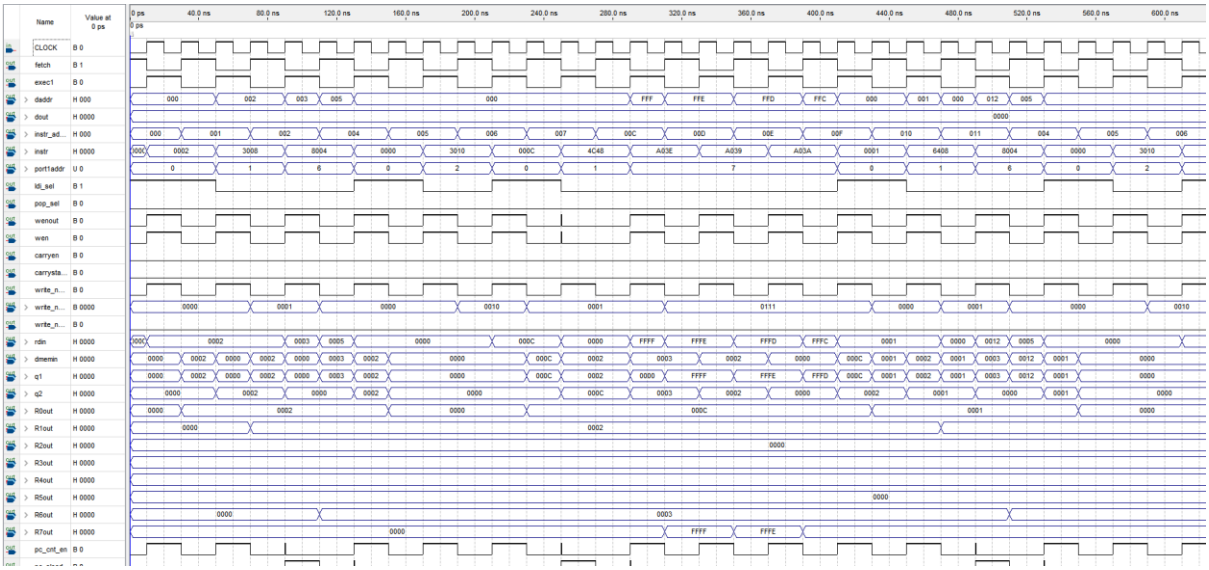
Slow 1200mV 85C Model Fmax Summary				
	Fmax	Restricted Fmax	Clock Name	Note
1	66.12 MHz	66.12 MHz	clk	

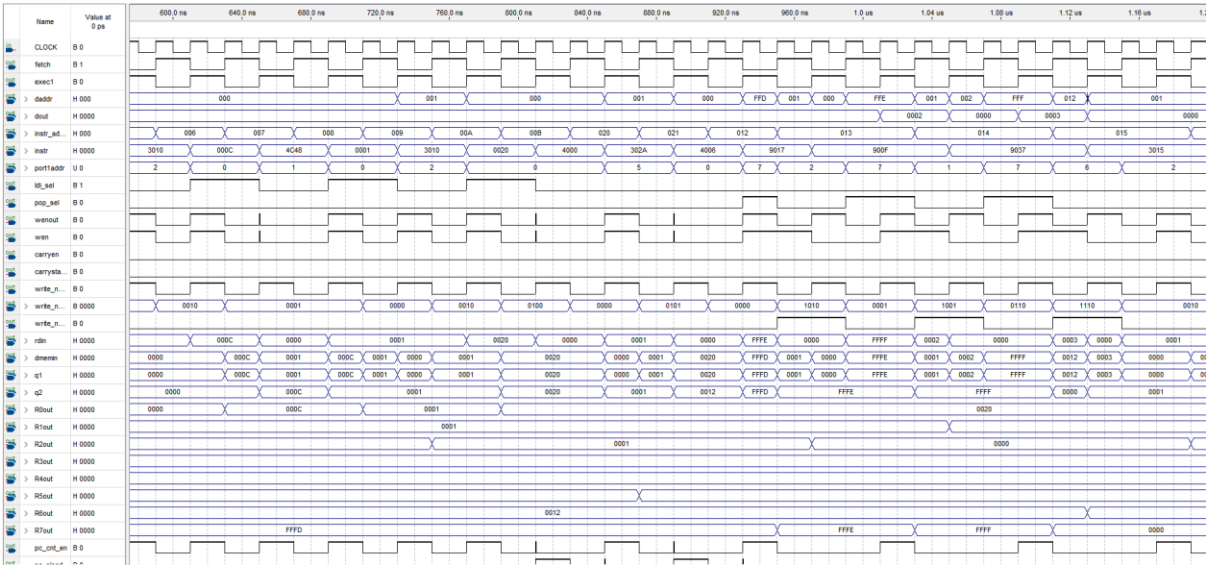
PowerPlay Power Analyzer Summary	
PowerPlay Power Analyzer Status	Successful - Thu Jun 11 18:06:48 2020
Quartus Prime Version	16.0.0 Build 211 04/27/2016 SJ Lite Edition
Revision Name	CPU
Top-level Entity Name	CPU_block_timing
Family	Cyclone IV E
Device	EP4CE6E22C6
Power Models	Final
Total Thermal Power Dissipation	133.94 mW
Core Dynamic Thermal Power Dissipation	61.56 mW
Core Static Thermal Power Dissipation	43.01 mW
I/O Thermal Power Dissipation	29.36 mW
Power Estimation Confidence	Low: user provided insufficient toggle rate data

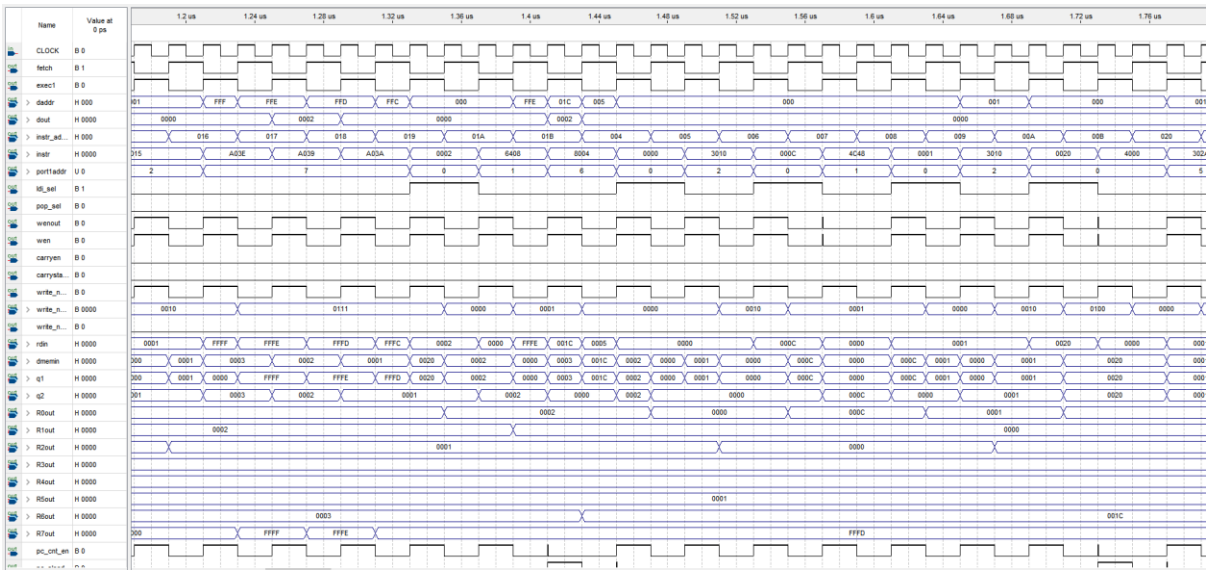
Appendix 10: Fibonacci results

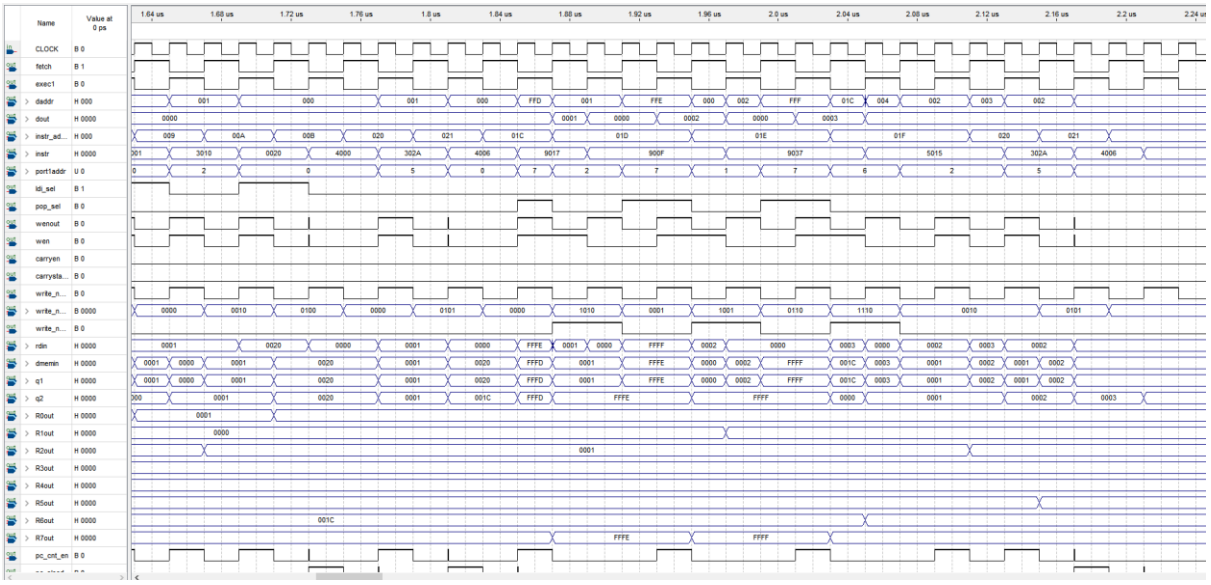
Fib(2) = 2

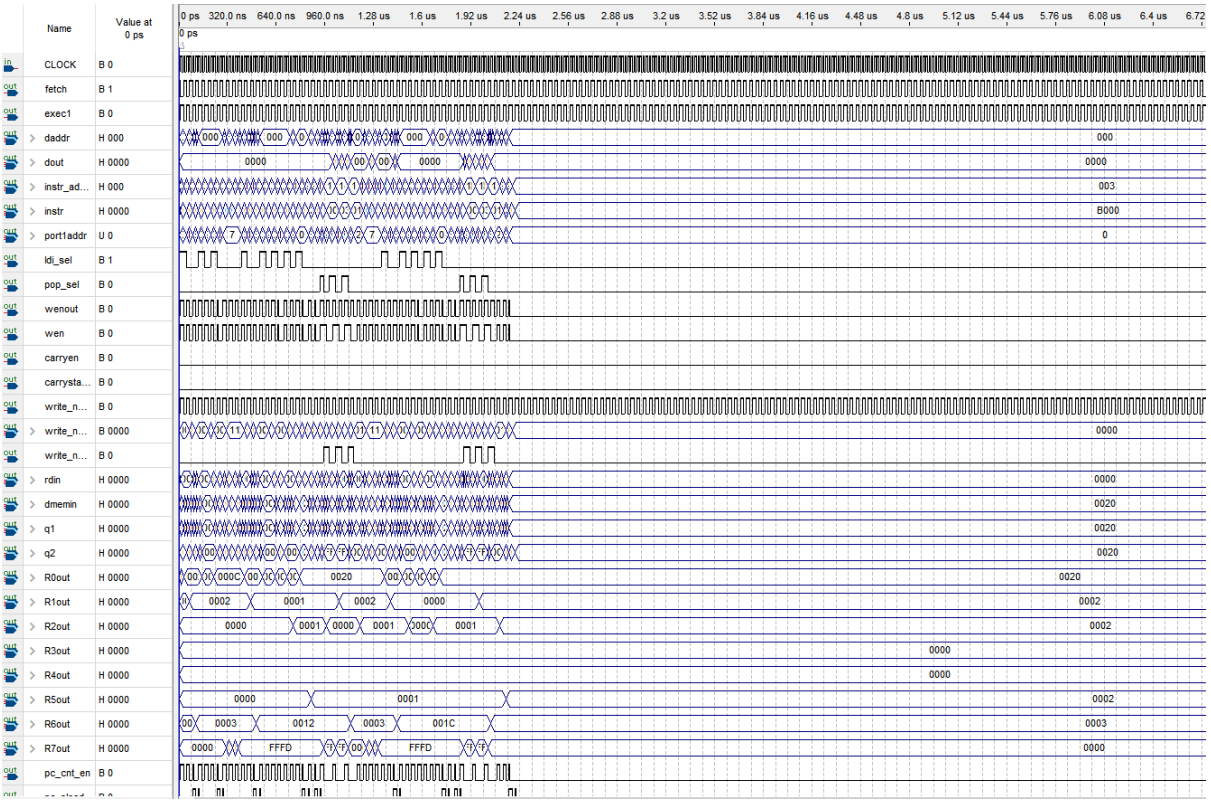
107 cycles
2.15 us



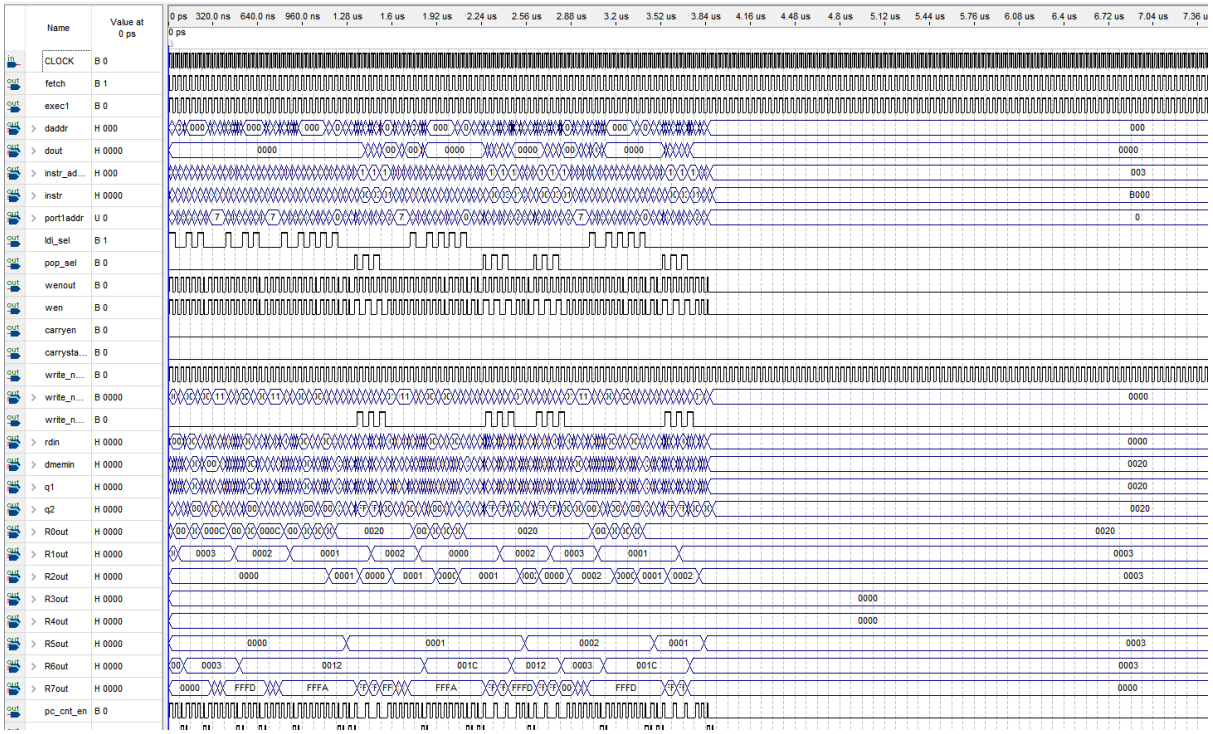








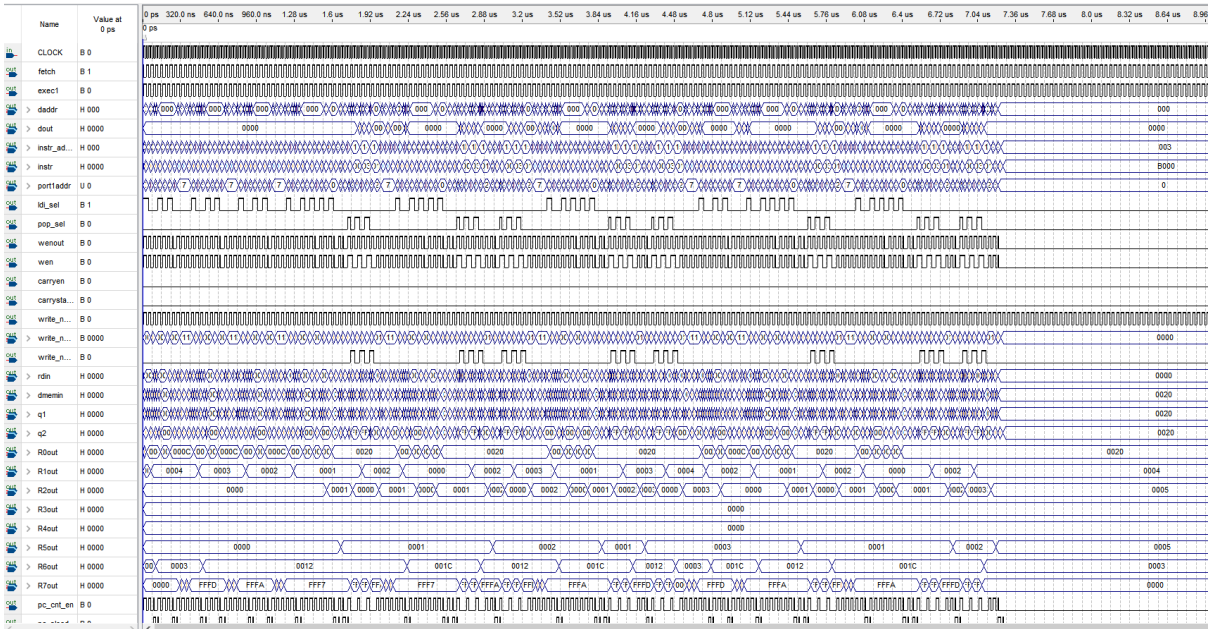
FIB(3) = 3
191 cycles
3.83us



FIB(4)=5

359 cycles

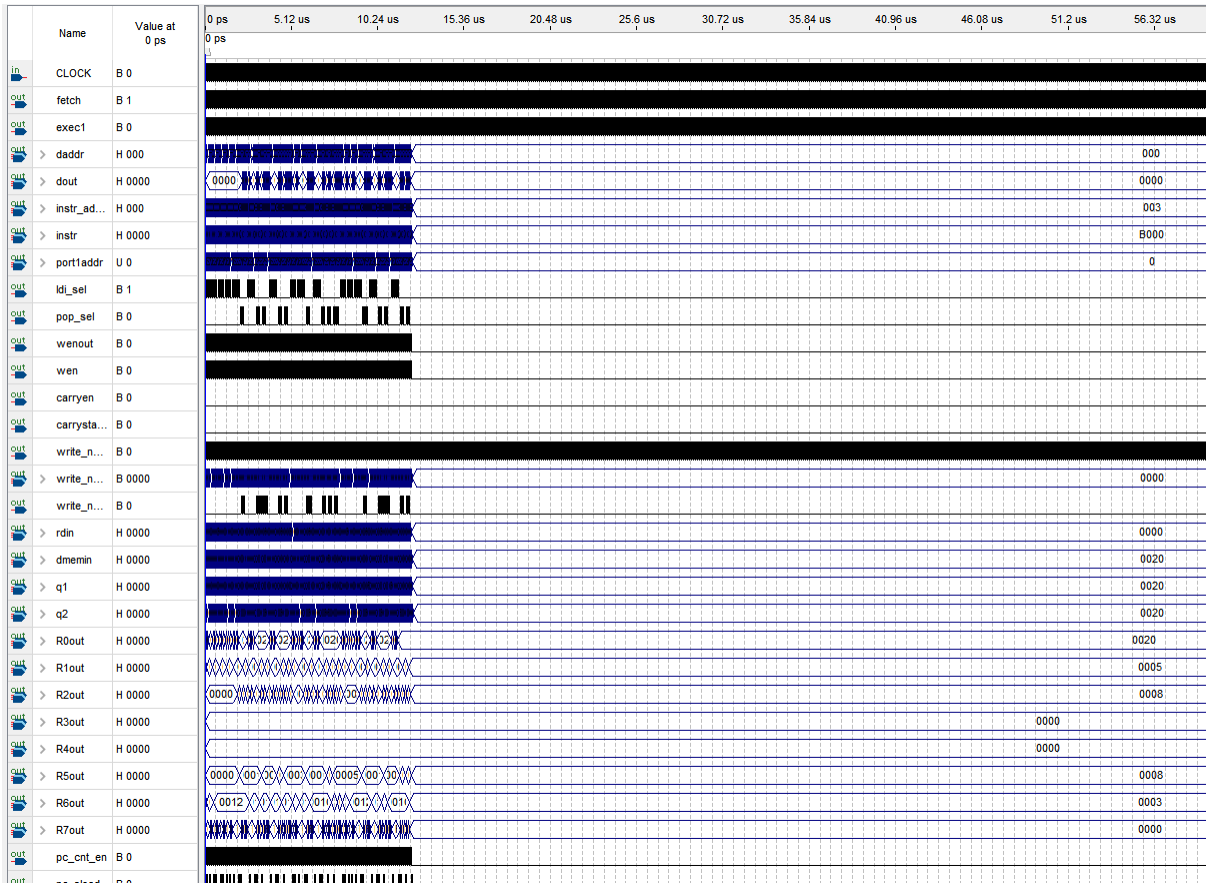
7.19 us



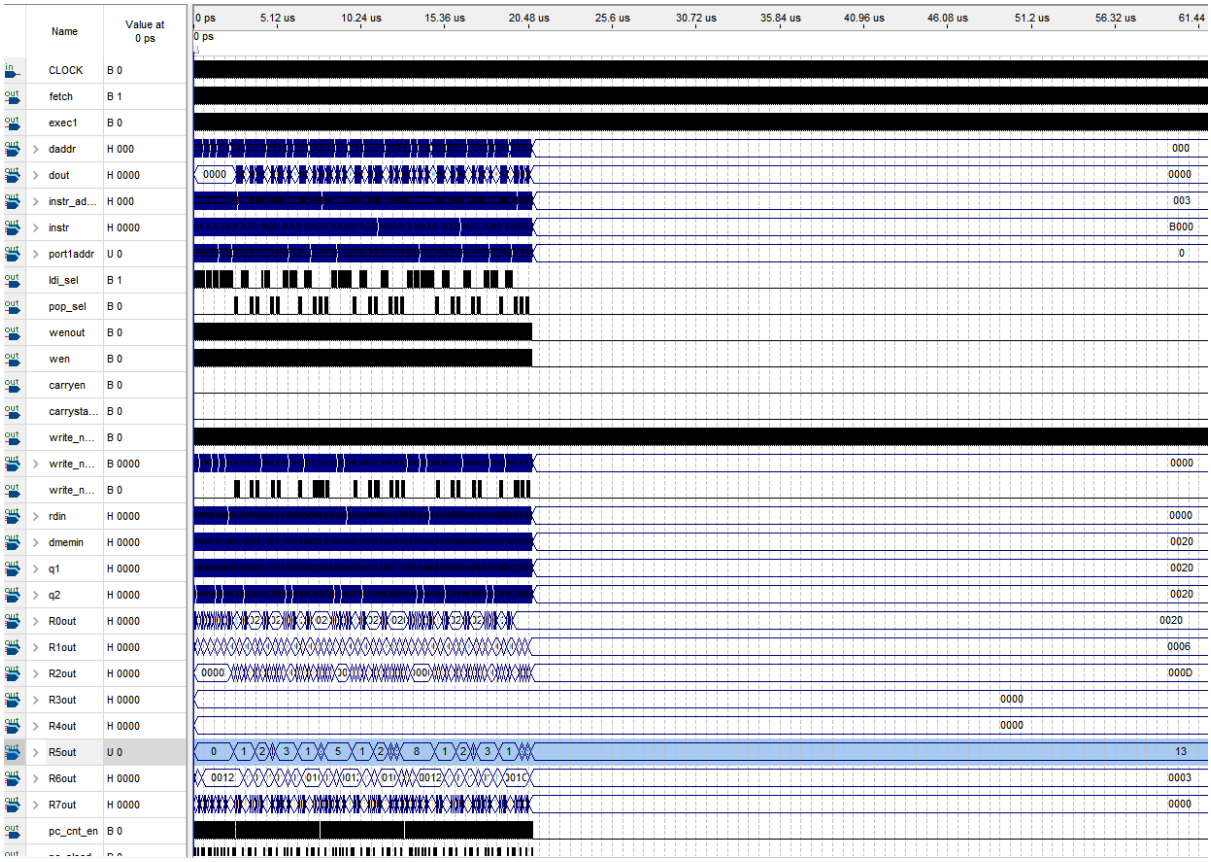
FIB(5)=8

611 cycles

12.23 us



FIB(6)=13
1031 cycles
20.63 us



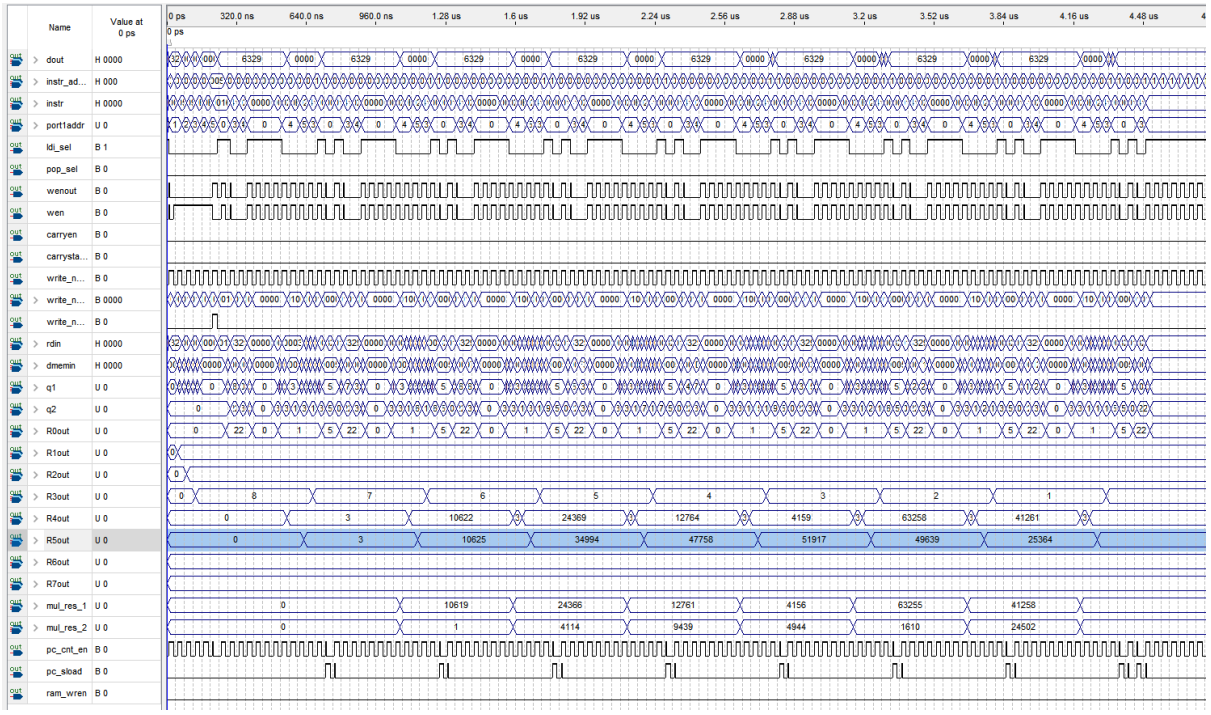
FIB(7)=21

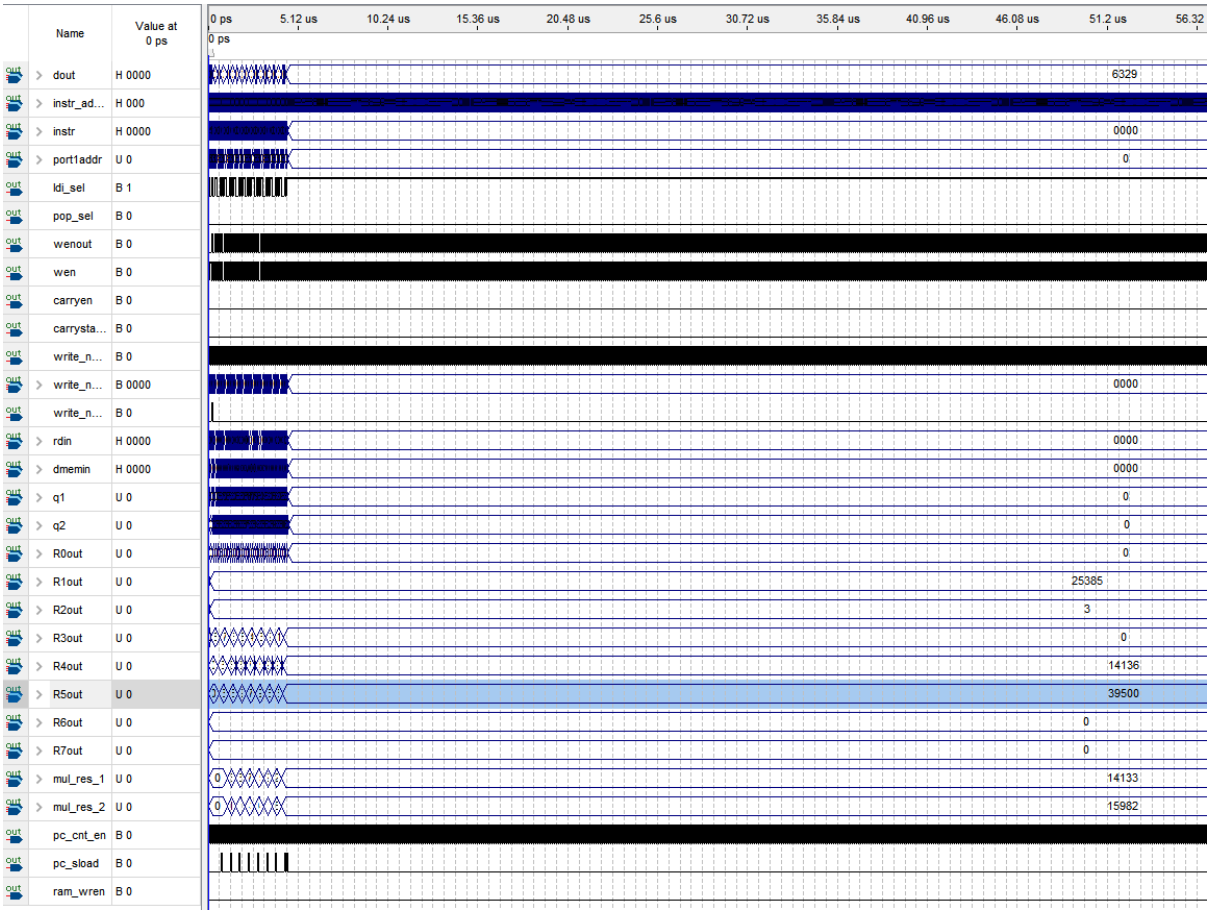
1703 cycles

34.07 us

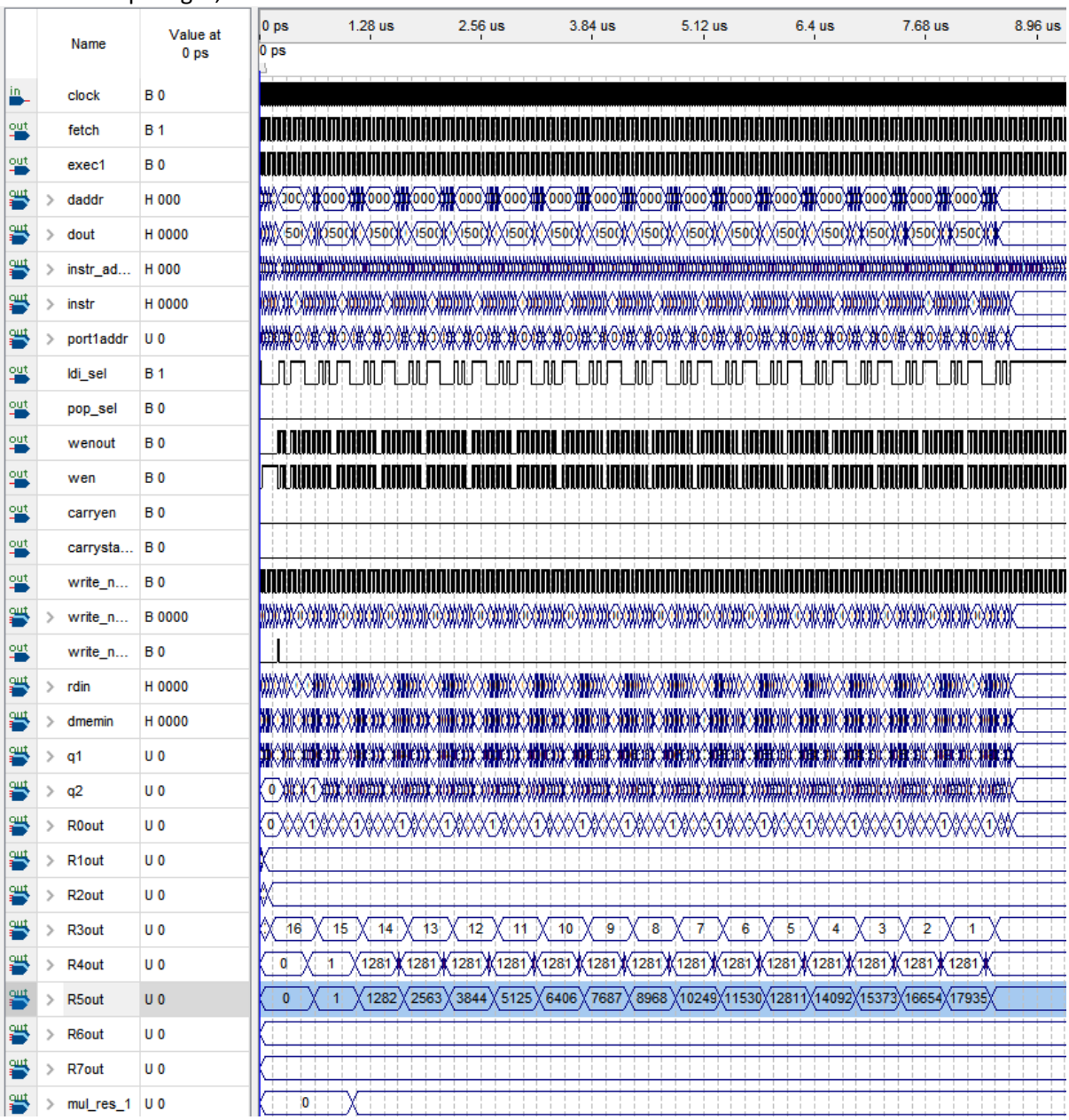
Appendix 11: Pseudo-random integer generator simulations

Using "typical" parameters in overview
A=25385=0x6329, b=3, n=8
213 cycles
Execution time: 4.27 us
Pattern: 0, 3, 10625, 34994, 47758, 51917, 49639, 25364, 39500 (does not seem to repeat)



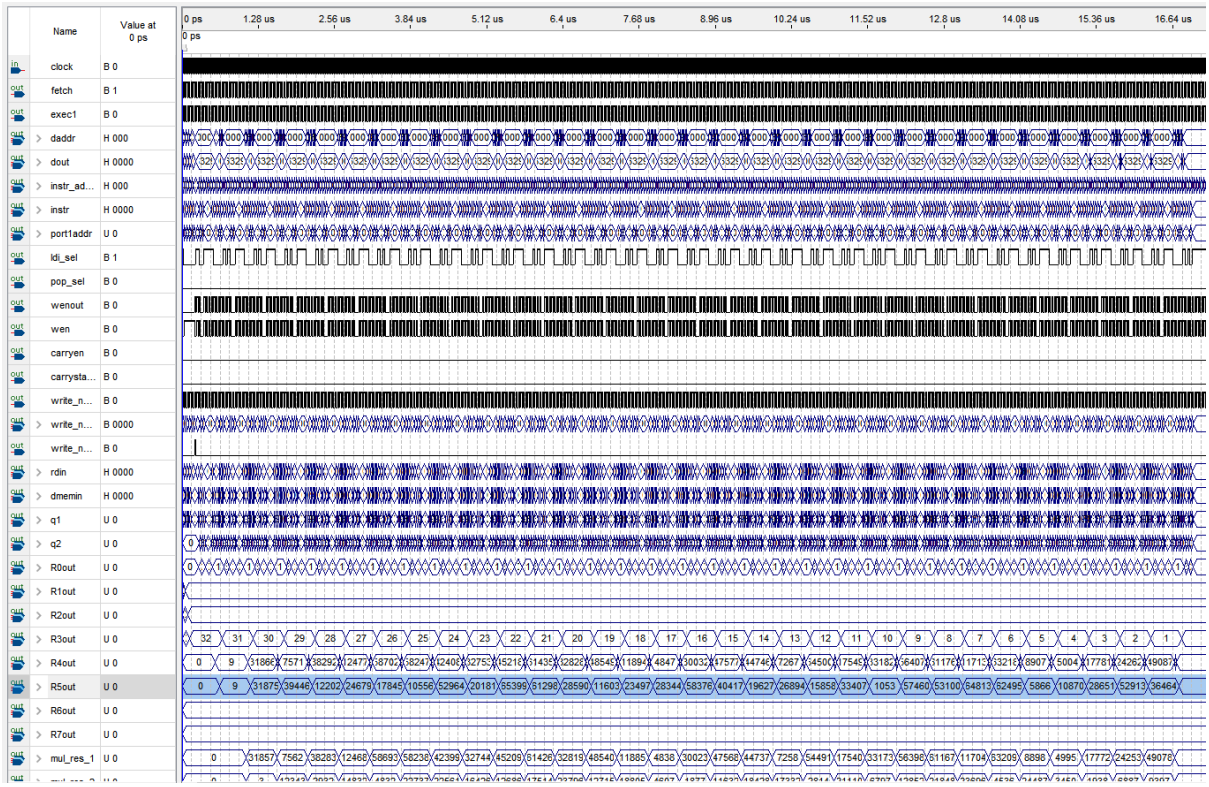


Using smaller values and longer loop, see if it repeats:
A=0x500, b=1, n=0x10
421 cycles
Doubled loop length, execution time doubled: 8.43 us



- Series still does not seem to repeat

Keeping big and longer loop:
0x6329, b=9, n=0x20
837 cycles
Execution time: 16.75 us

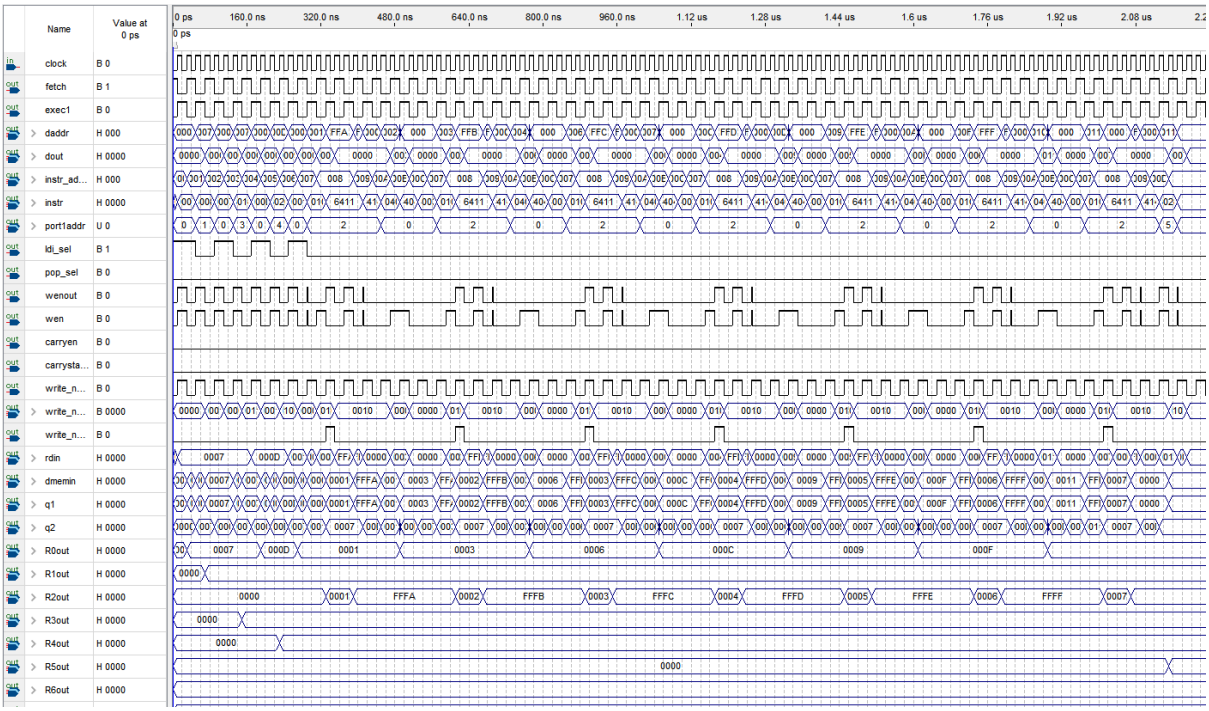


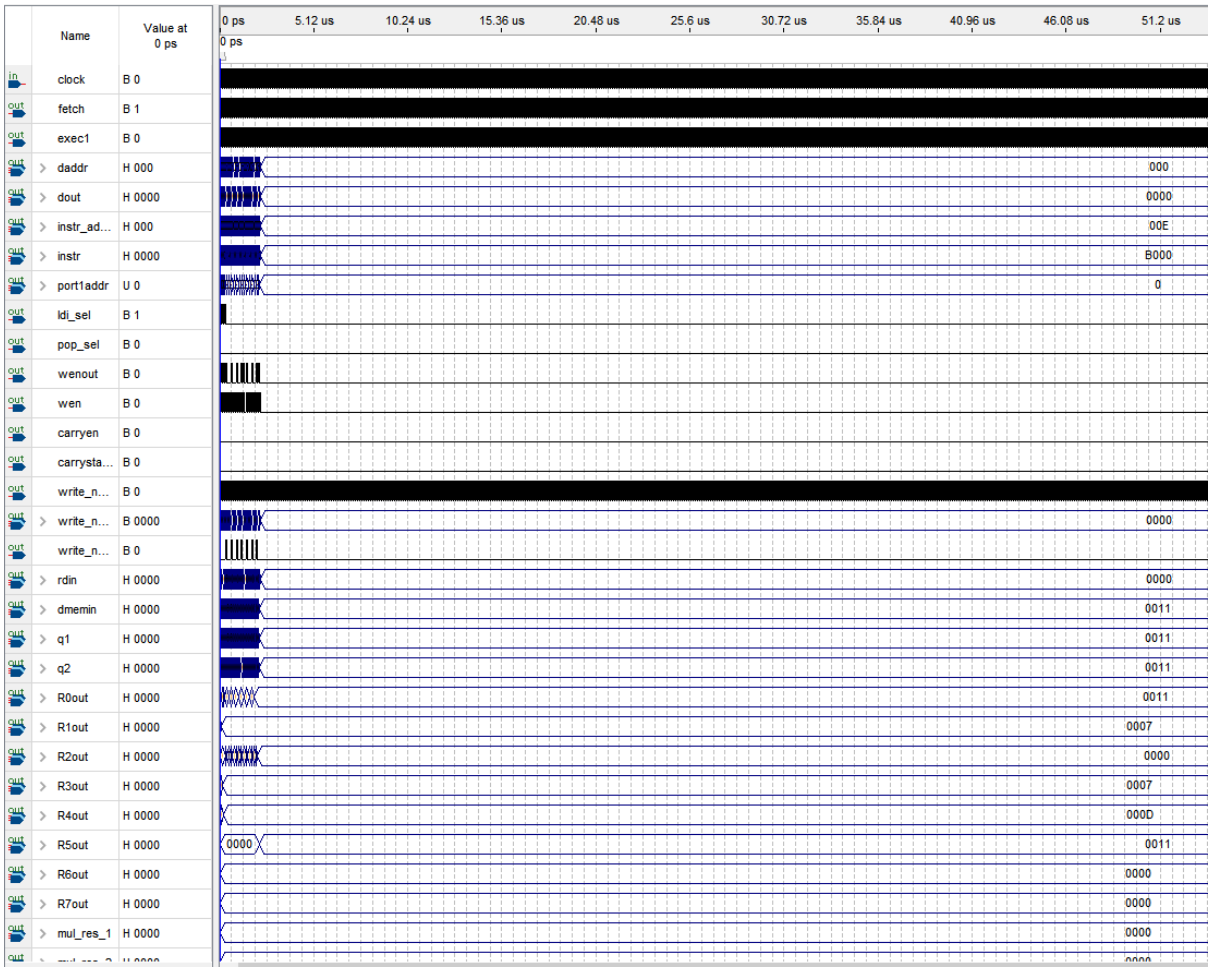
Appendix 12: "Find value in list" Benchmark results

Length 7:

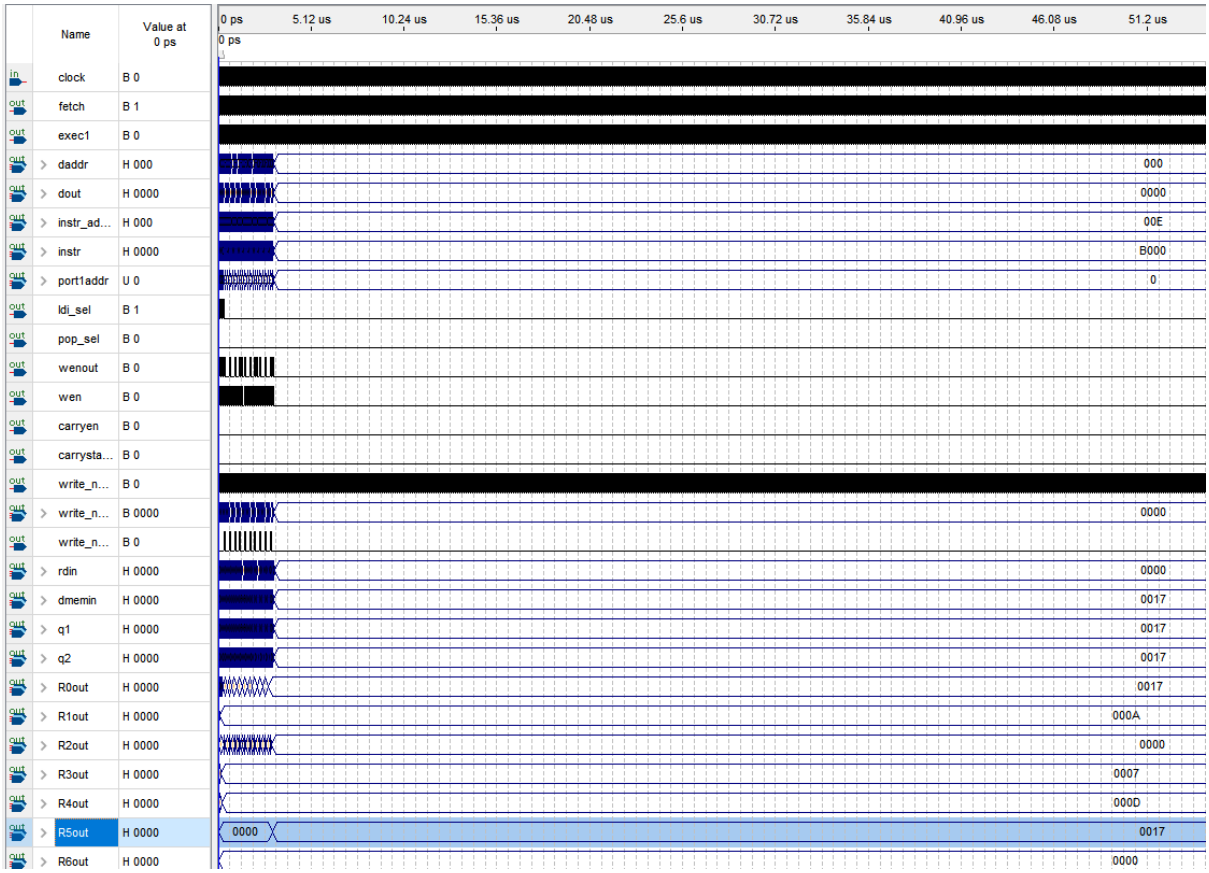
107 cycles

2.15 us





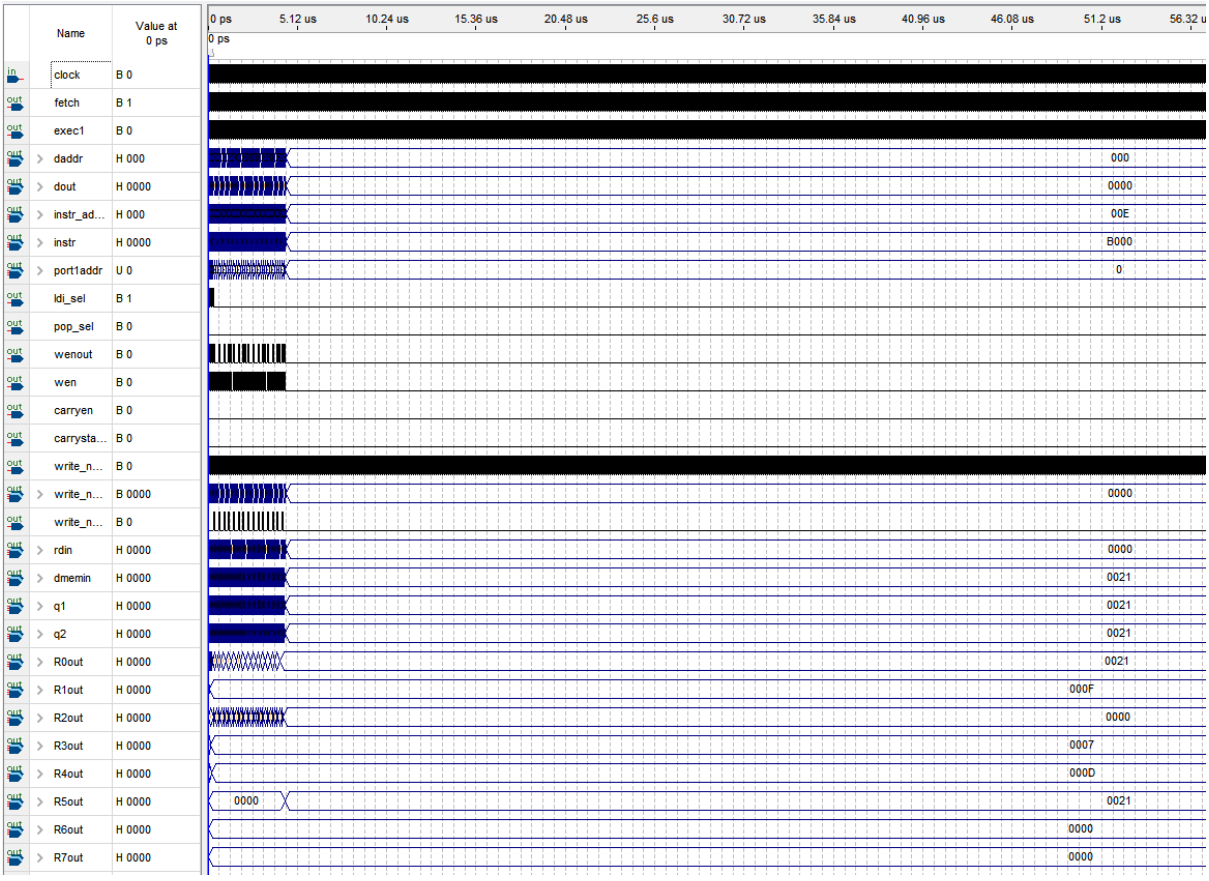
Length 10 (typical):
149 cycles
2.99 us



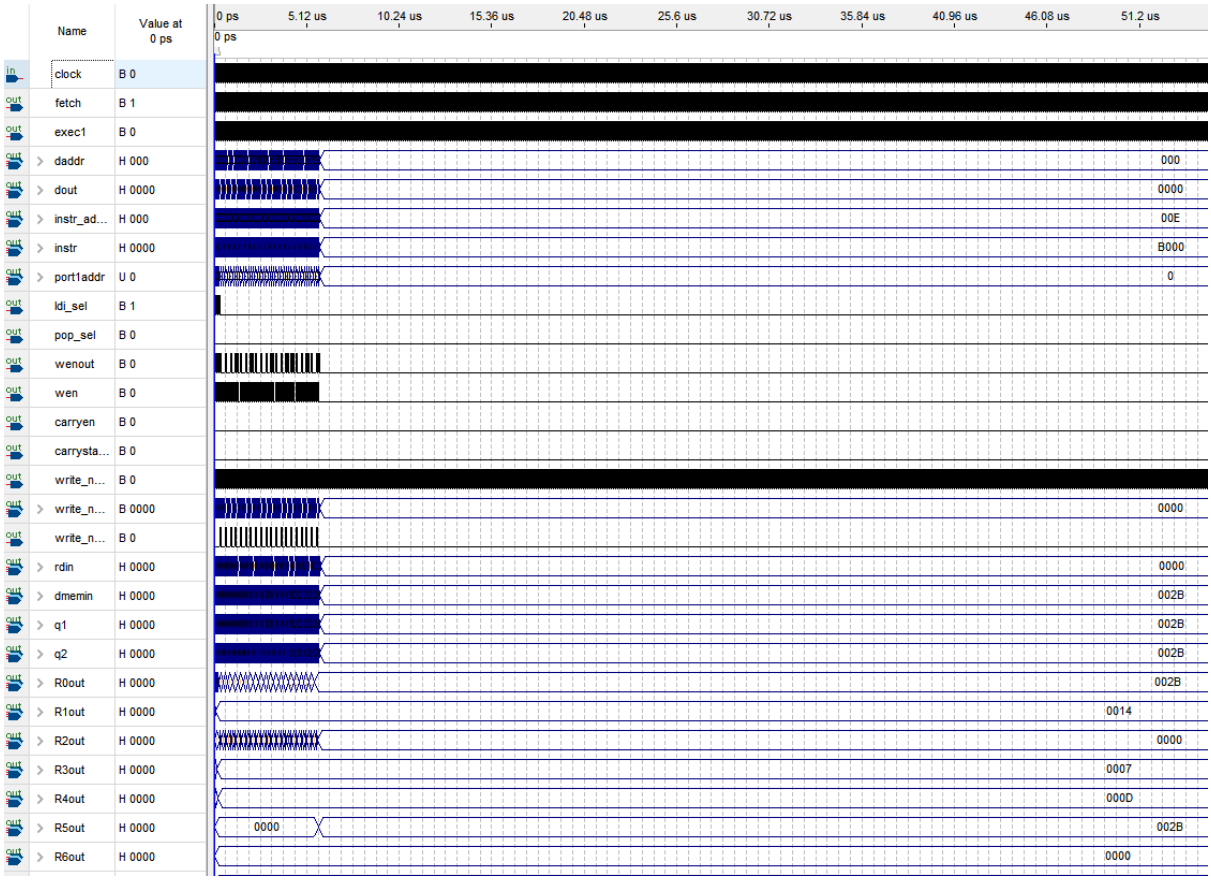
Length 15:

219 cycles

4.39 us



Length 20:
289 cycles
5.79 us

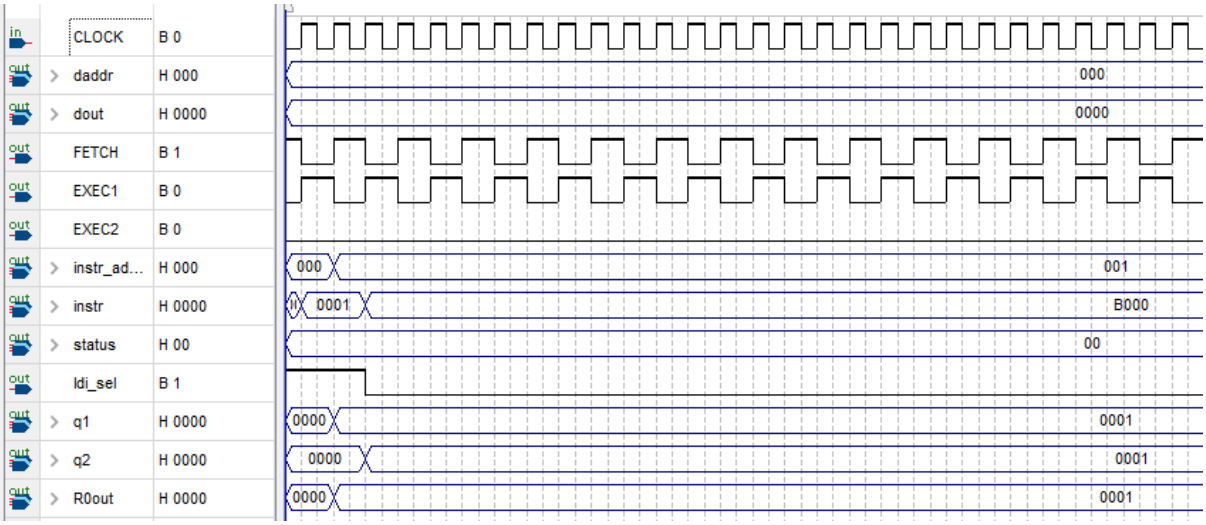


Appendix 13: Instruction testing process

LDI test

Instruction mem:

LDI 1	0001
STP	B000



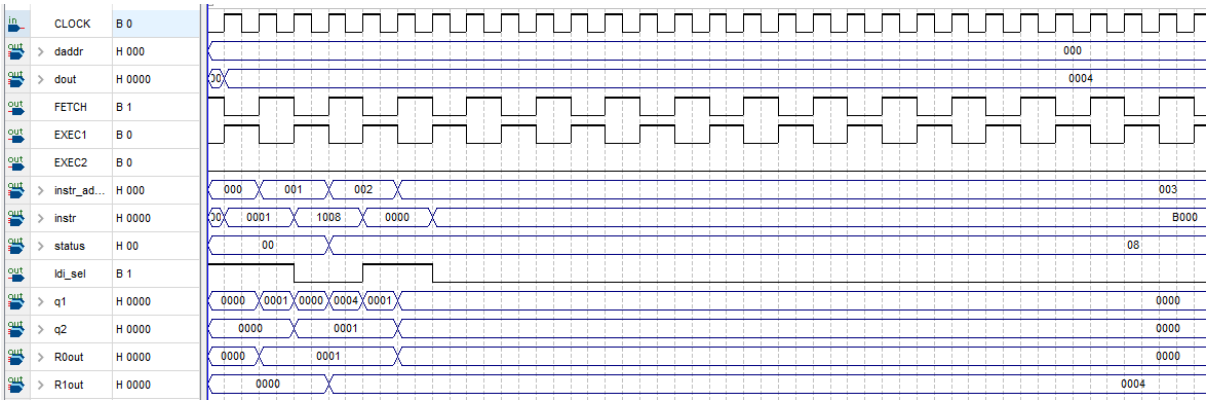
LDR

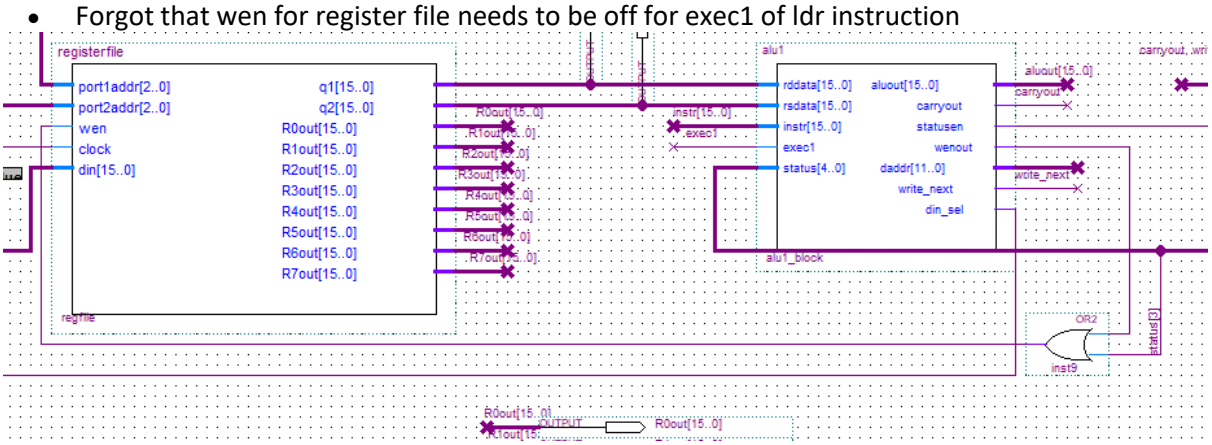
LDR (no offset) test:

Data memory:
0x0004
0x00AB

Instruction mem:

LDI 0x1	0001
LDR 0 0 R1 R0	1008
LDI 0x0	0000
STP	B000

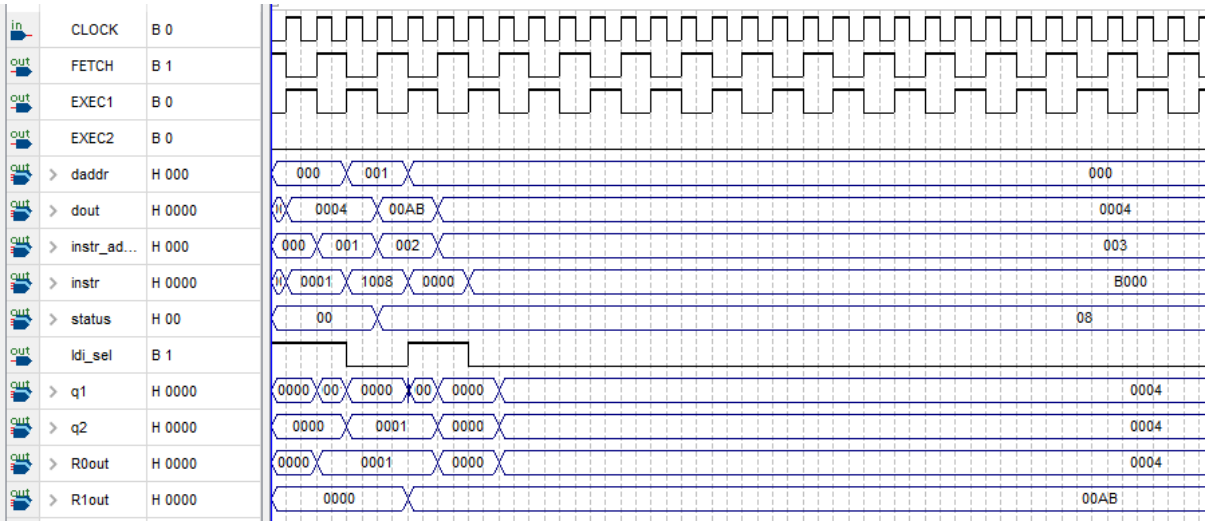
- Note - does not work yet - implemented later
  - 

  - Was loading 0x4 into R1
  - Forgot that wen for register file needs to be off for exec1 of ldr instruction
  - 

  - Removed ldi and ldmfd from wen and added the OR gate so that it writes when the write_next bit in status bus is active even when wenout is off

Another issue:

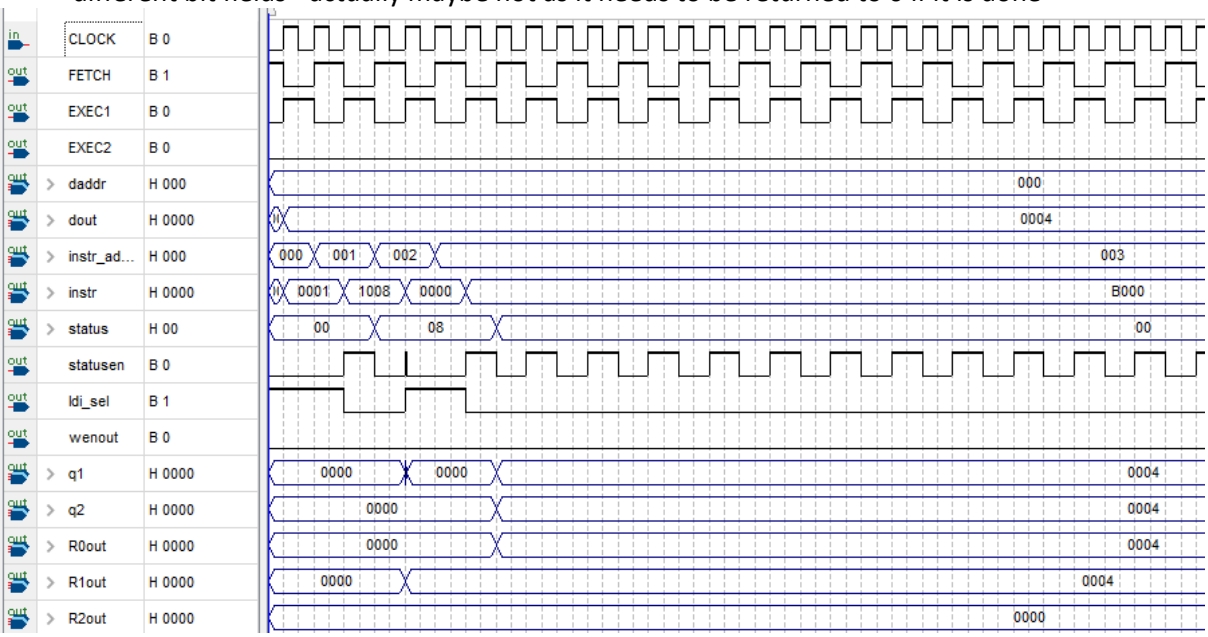
- Daddr remains at 0
- Daddr pin was not named



- During STP, R0 is written to since the initial values in memory is 0000, which is LDI 0000, which loads R0 with 0000
- Need to add a STP bit in status register to say that the program has stopped and wenout in ALU should be off
- After doing this, wenout was stuck at 0
- This is because stp_en is initialised to 1
    - Should expect 0 as register is initialised to 0, so initial status[5] should be 0

Status register:

- Do we need to have separate registers so that I can individually enable writing to the different bit fields - actually maybe not as it needs to be returned to 0 if it is done

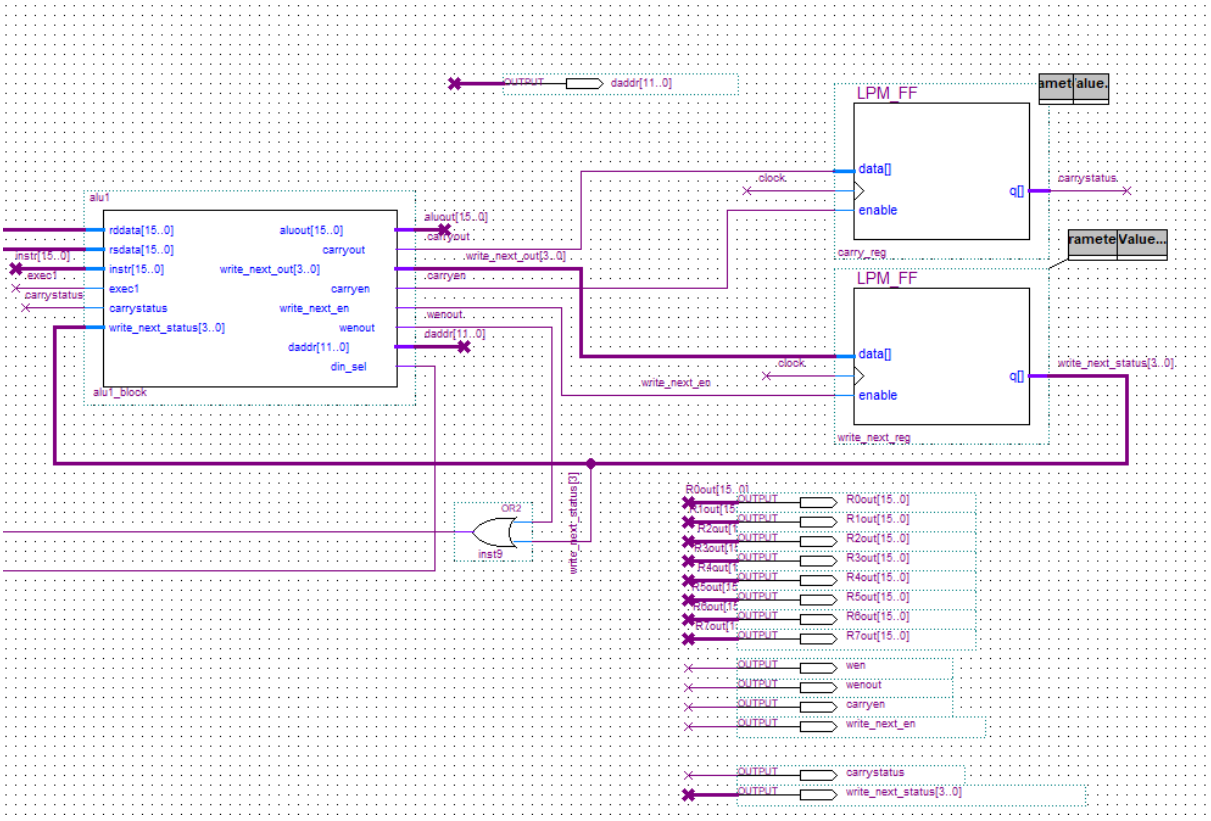


- Could not see stp_en as bit size was not right in simulation
- It is always at 1

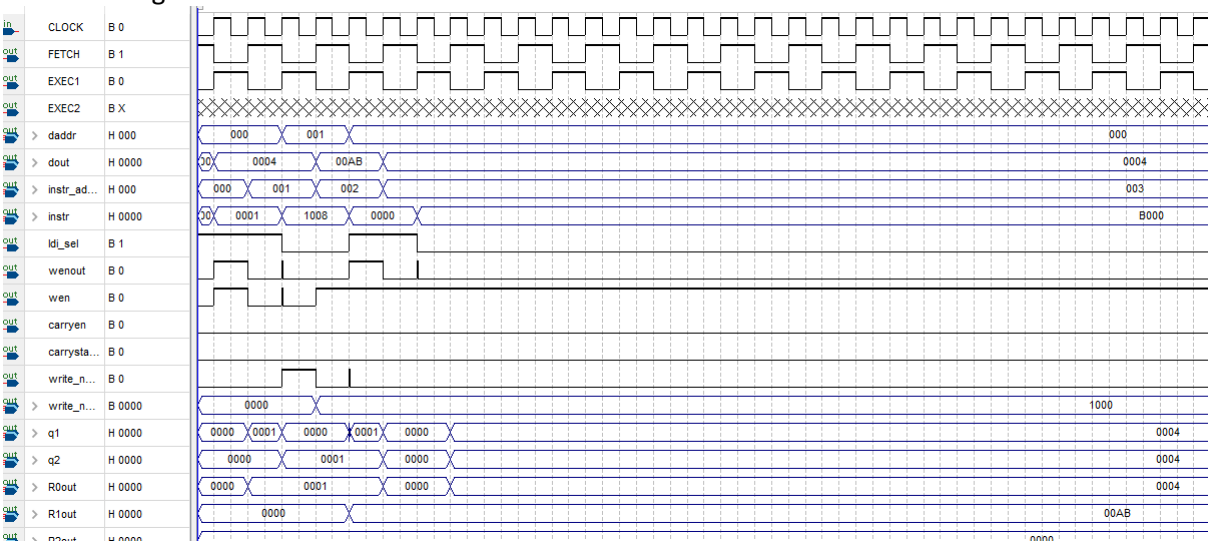
Stp_en is actually not needed as instr stays ta B000, for some reason thought it went to 0000

- But wenout is off while R0 is changing???
- Status[3] is on, which turns on wenout
- Need it to turn off one cycle afterwards

- Don’t need stp_en stuff and we need to separate status register to cin register and write_next register



Simulation again:

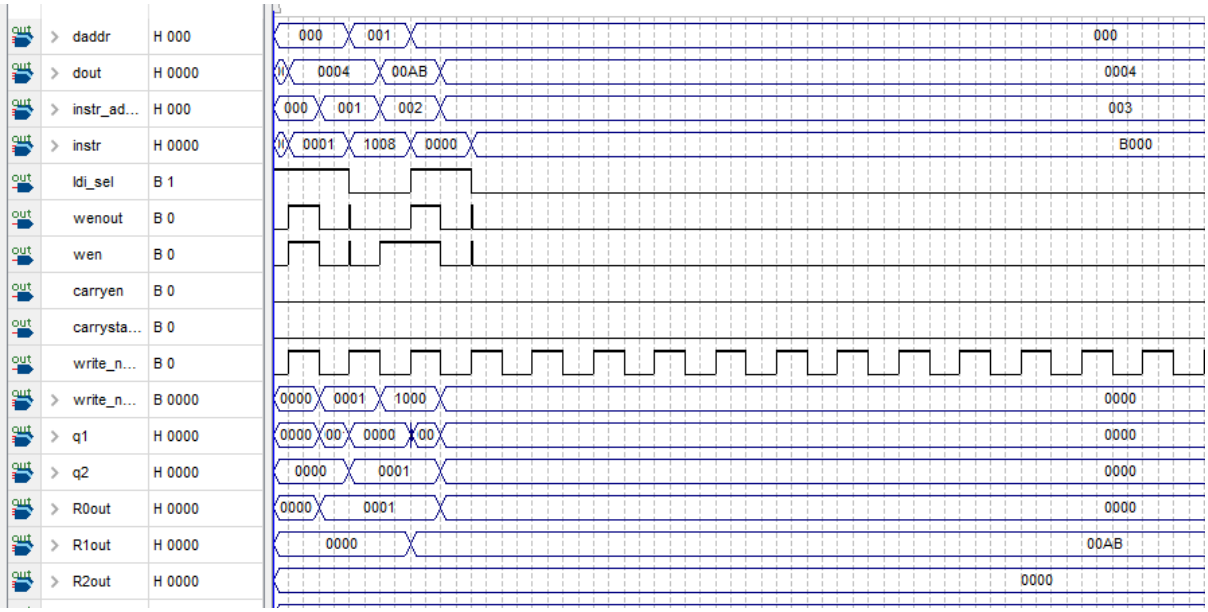


- Same problem, wen is on with stp as write_next_out[3] is on and is not switched off after loading in next cycle

Issue was that write_next_en was only on for ldi and ldmfd, when it should always be on during exec1, so that it can actually go back to 0

```
// write_next_en enables writing to write_next register - needs to update for every instruction during exec1 so that it returns to 0
assign write_next_en = exec1;
```

Simulation working:



- Realised this doesn't work as it will be pipelined - can't actually LDI after LDR - can't write to registers as there is a conflict for what to have as port1addr

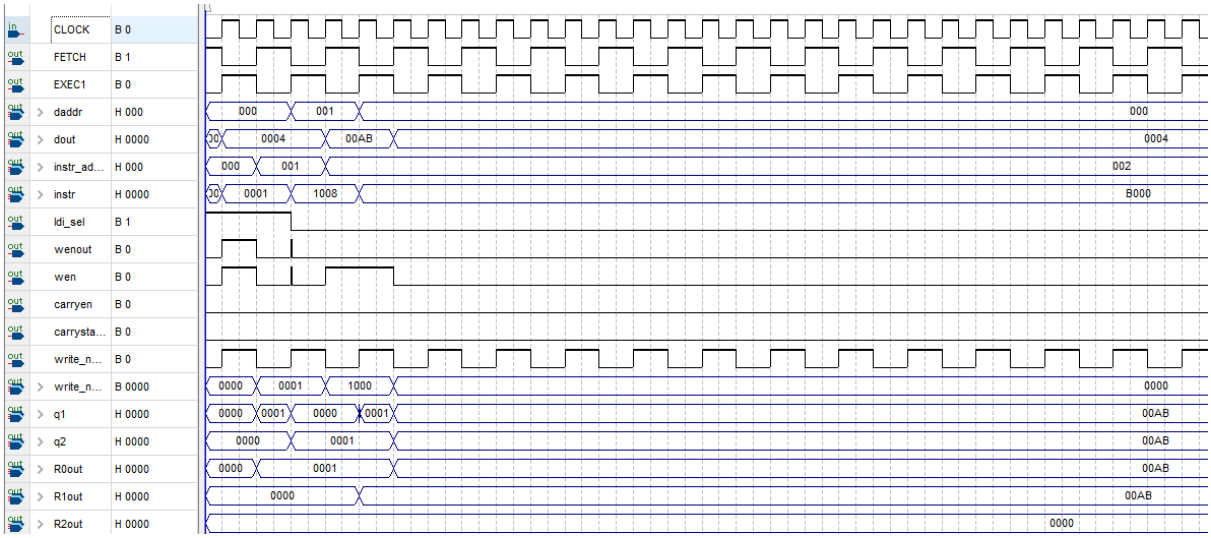
Test STP right after LDR

Data memory:
0x0004
0x00AB

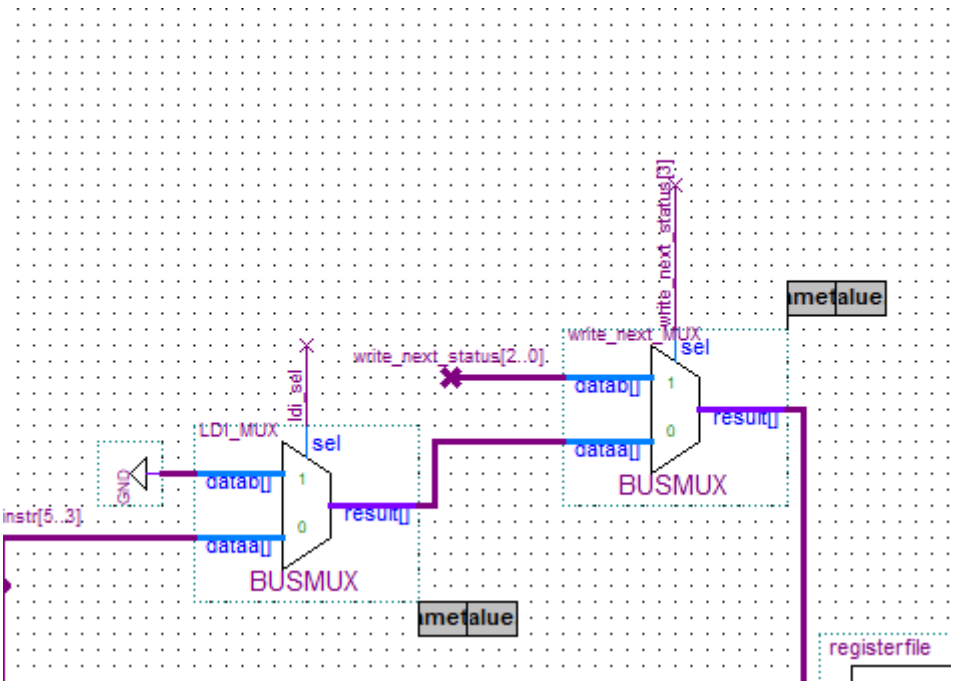
Instruction mem:

LDI 0x1	0001
LDR 0 0 R1 R0	1008
STP	B000

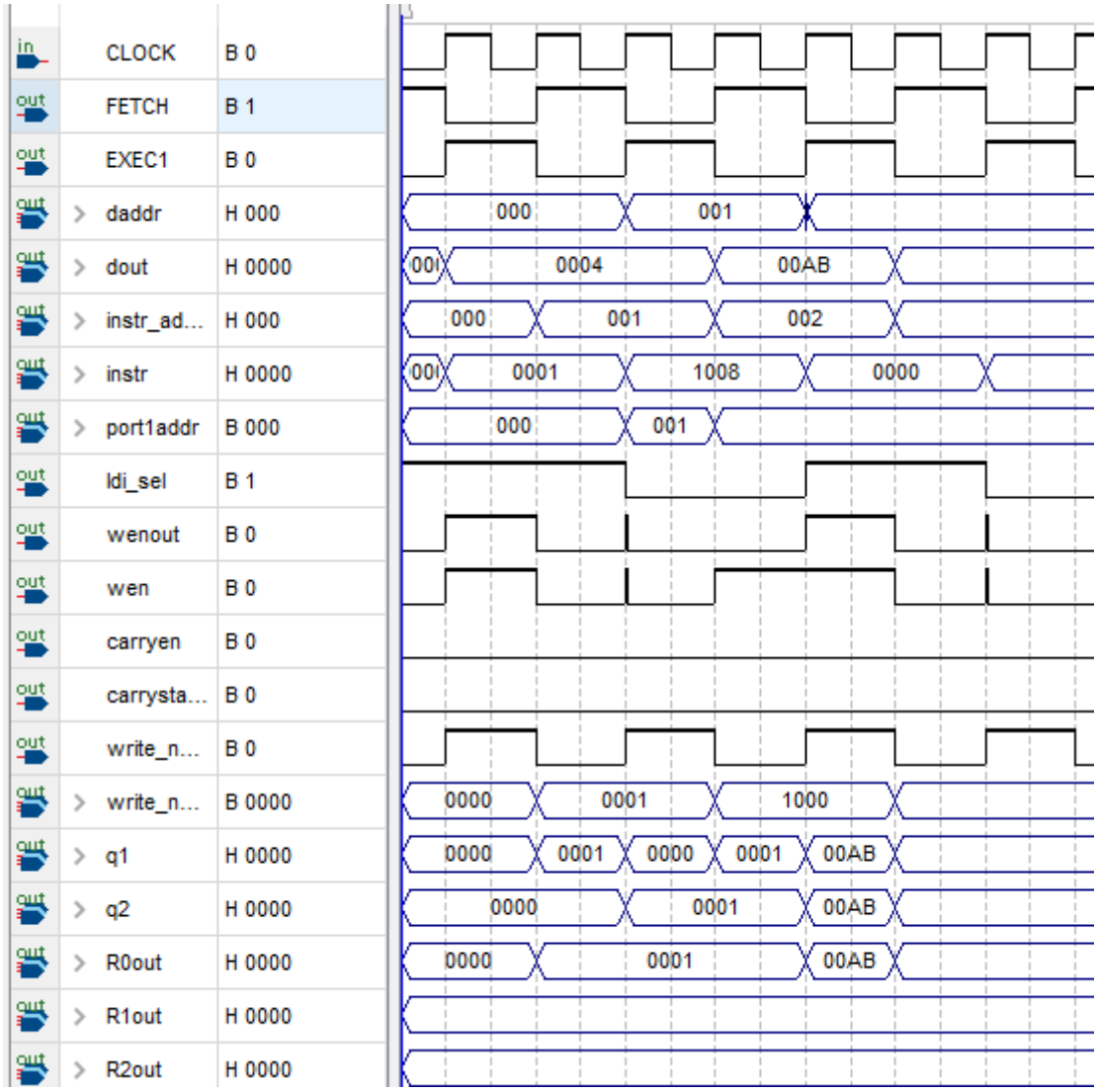
Issue:



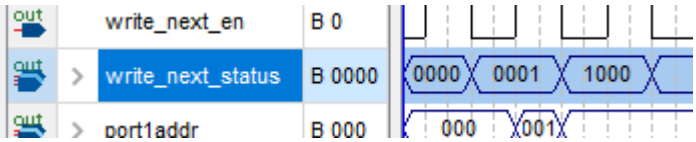
- R0 is changing since wen is enabled while daddr is 0 during STP exec1
- Port1addr needs to come from status register, not instruction word - forgot to actually take status register address and put it into port1addr -> enabled if status[3] is enabled
    - Added MUX:



Old simulation:



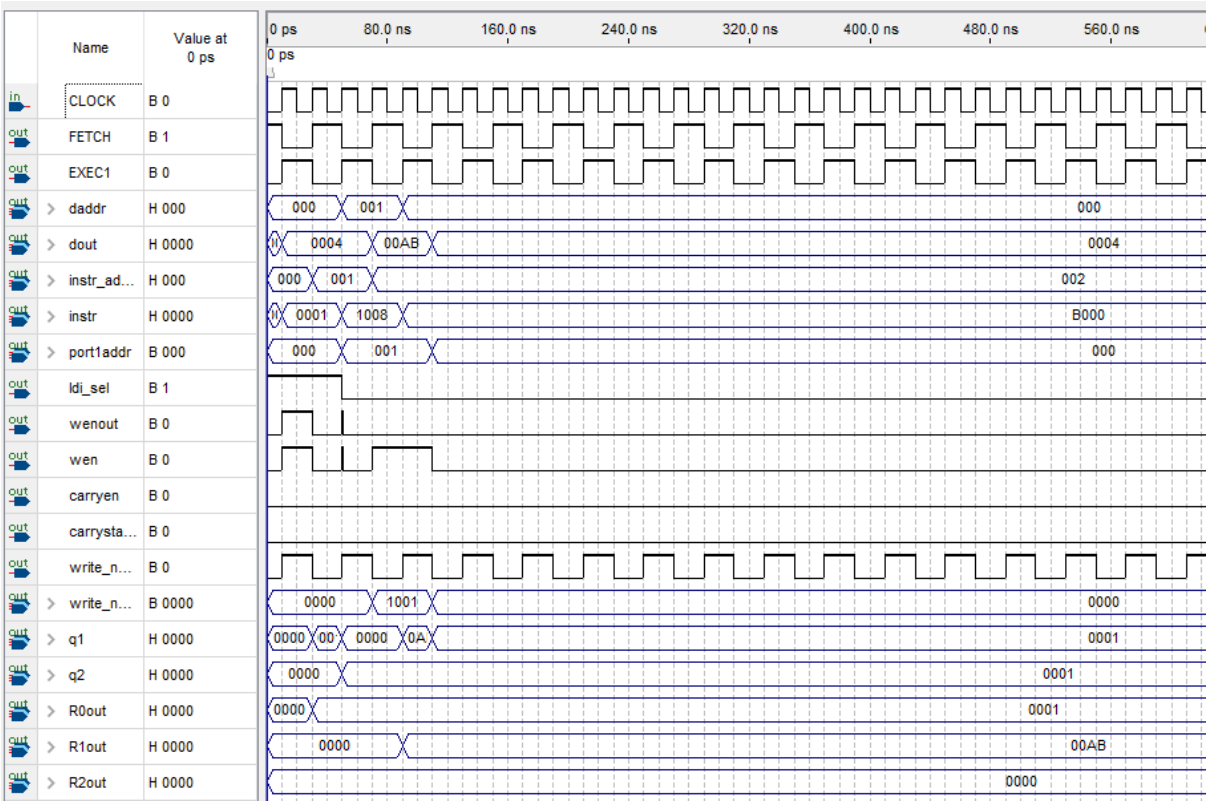
- Port1addr is 1 during ldr instruction rather than one cycle after when dout has updated



- Status is not storing 1001
  - Storing 1000 so it loads to R0 (sees register address as 0)
- It was taking Rs into write_next rather than Rd address, corrected:

```
// Output to write_next register
assign write_next_out = {write_next_flag, instr[5:3]};
```

Working;



- R1 loaded with 0x00AB or mem[1] and R0 remains at 1

Test two LDR instructions after each other

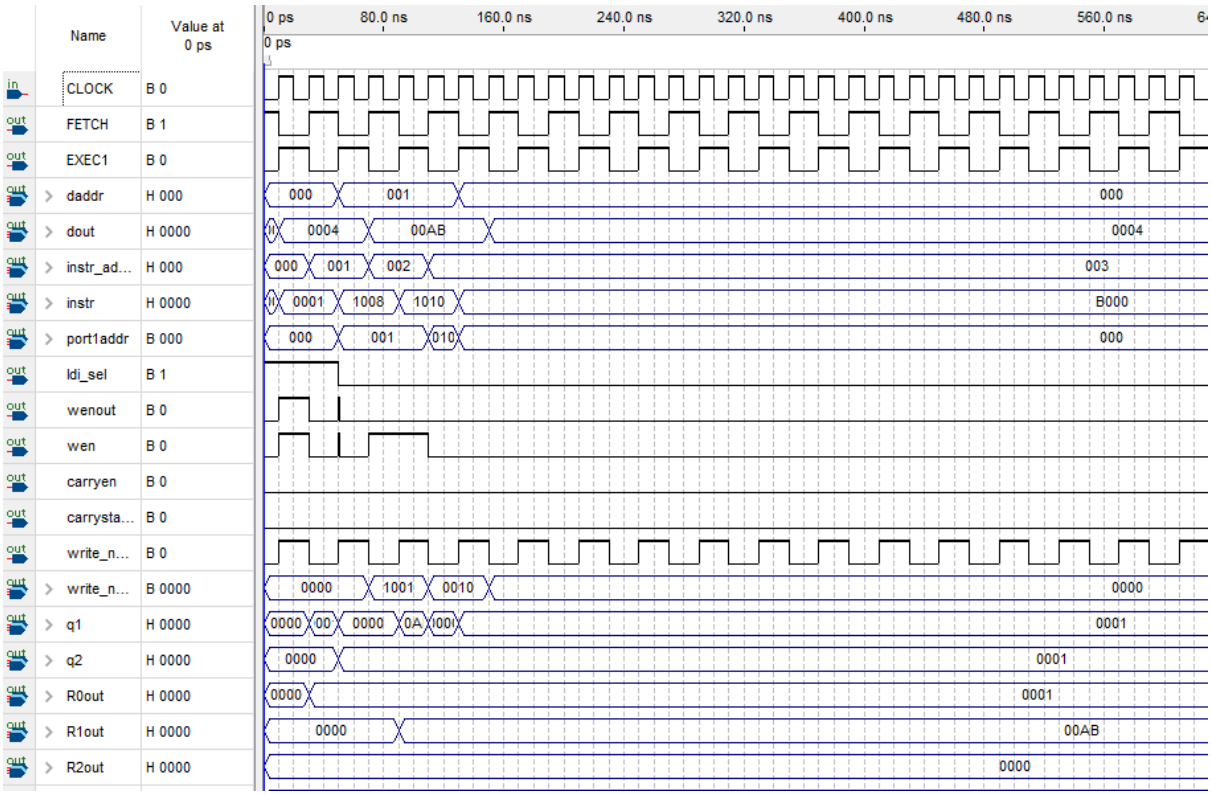
- Works since the first cycle of the second LDR does not write to registers

Data memory:
0x0004
0x00AB

Instruction mem:

LDI 0x1	0001
LDR 0 0 R1 R0	1008
LDR 0 0 R2 R0	1010
STP	B000

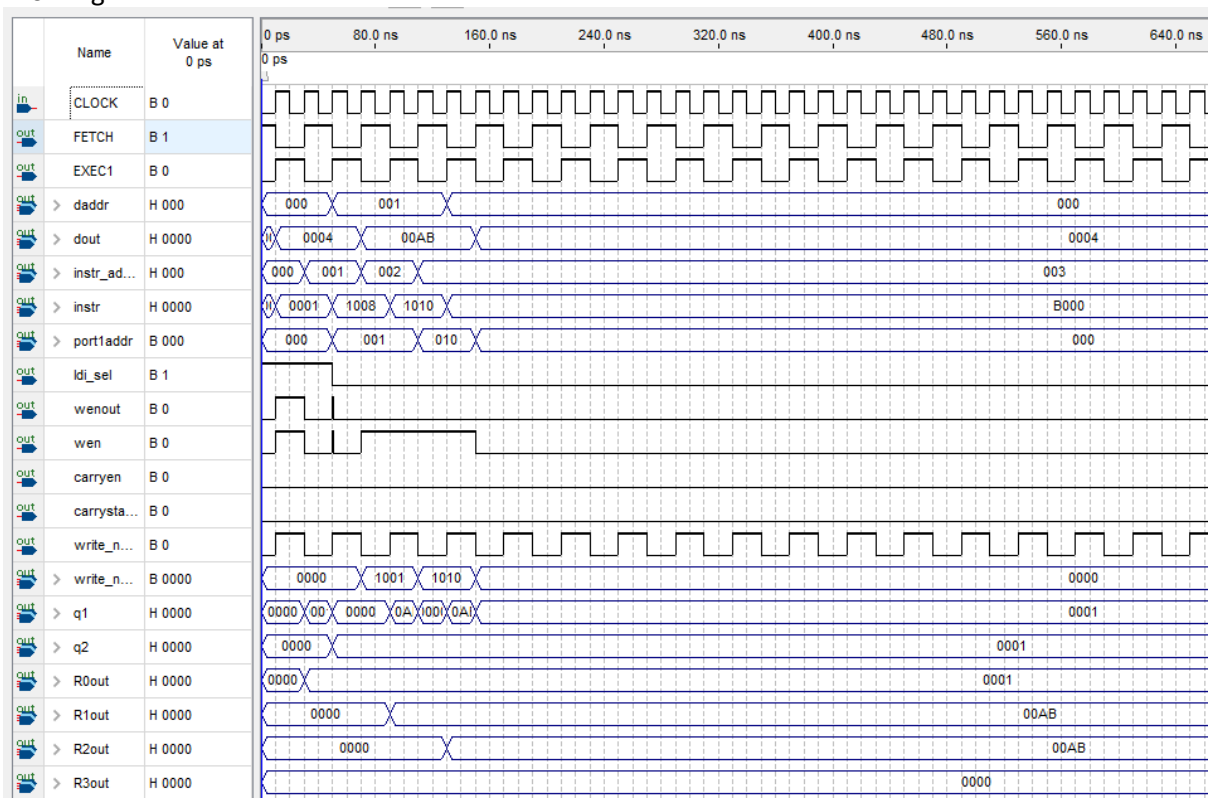
Issue:



- If the next instruction is an LDR, need to make sure that it does not reset write_next_flag to 0 as seen above

```
// write_next tells next instruction that the data that is now at dout can be written into the Rs of the previous instruction (for load instructions)
// if write_next_flag is already 1, set to 0 (if it is not another load), otherwise set to 1 if it is an ldr or ldmfd instruction
assign write_next_flag = (write_next_status[3] & ~(ldr|ldmfd)) ? 0 : (ldr | ldmfd);
```

Working:



Test multiple LDR instructions

Data memory:

0x0004

0x0002

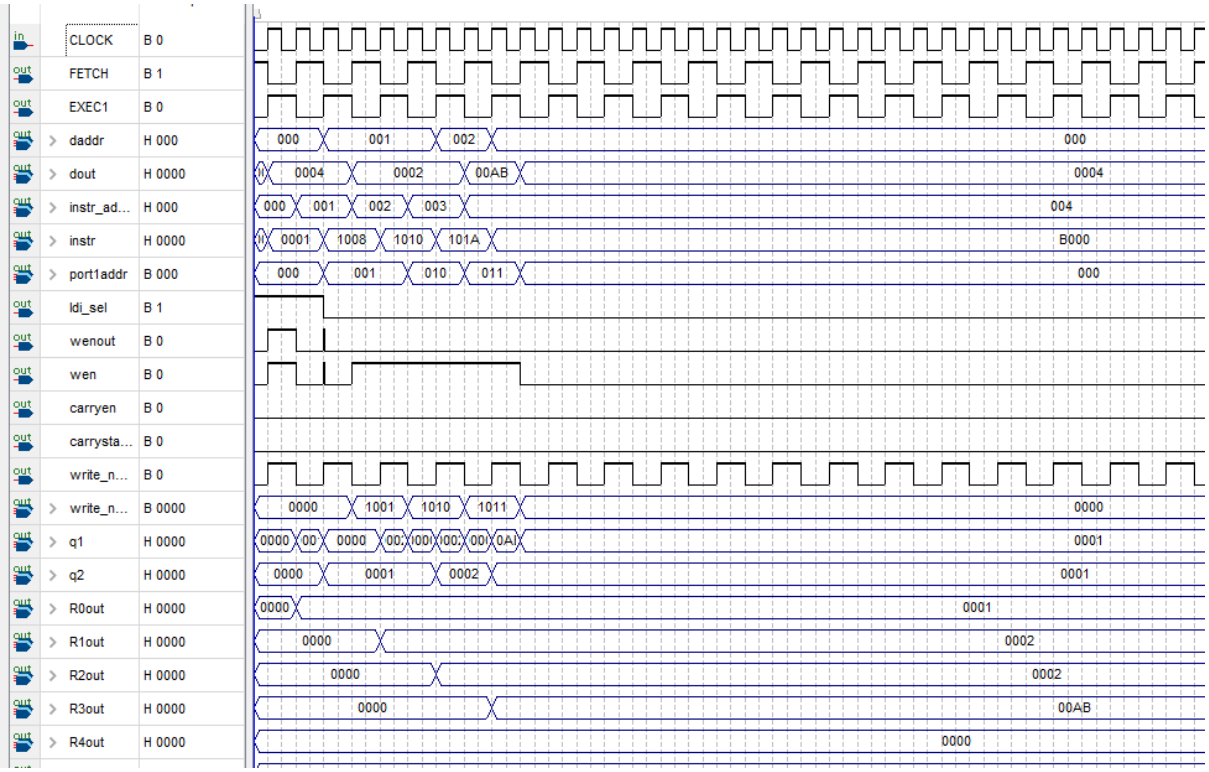
0x00AB

Instruction mem:

LDI 0x1	0001
LDR 0 0 R1 R0	1008
LDR 0 0 R2 R0	1010
LDR 0 0 R3 R2	101A
STP	B000

0001 1008 1010 101A B000

Works:



Test positive offset LDR

Data memory:

0x0001

0x002

0x003

0x006

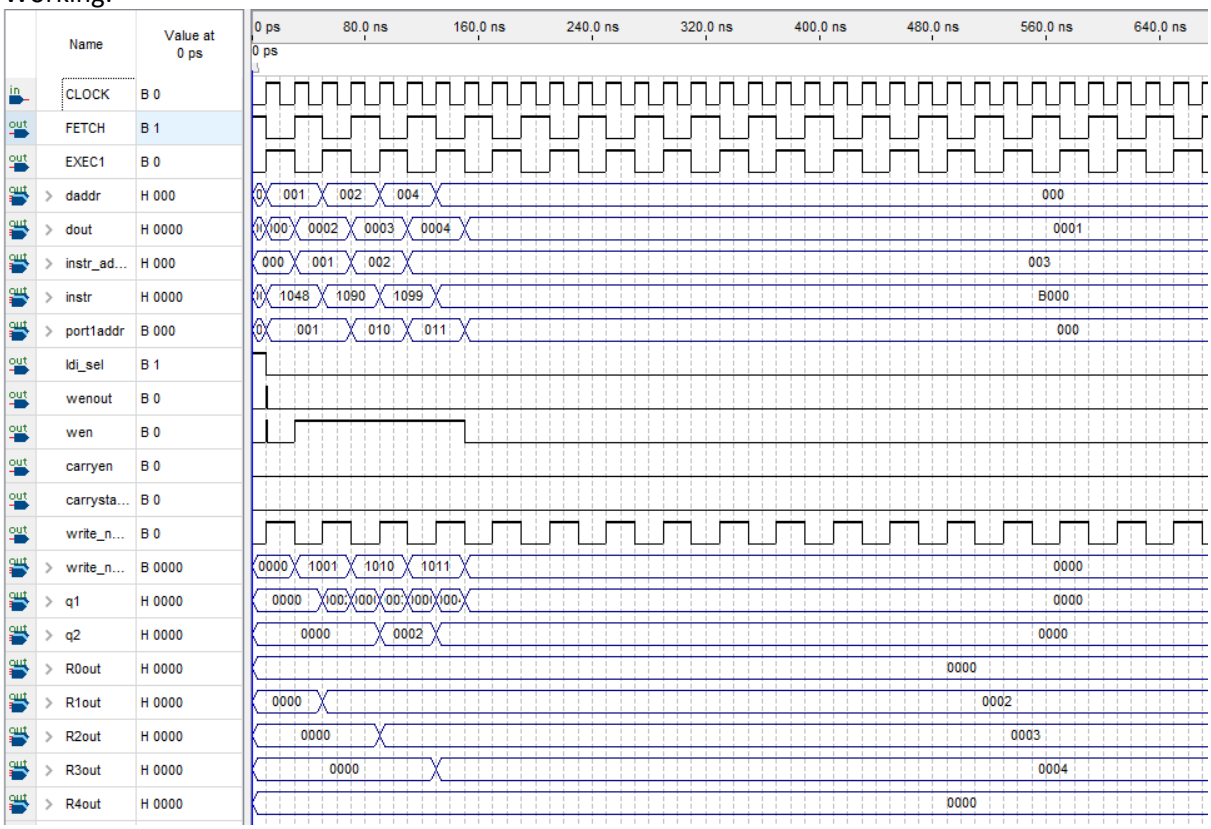
0x004

Instruction mem:

LDR 0 1 R1 R0	1048

LDR 0 2 R2 R0	1090
LDR 0 2 R3 R1	1099
STP	B000

Working:



Test negative offset LDR

Data memory:

0x0002

0x003

0x004

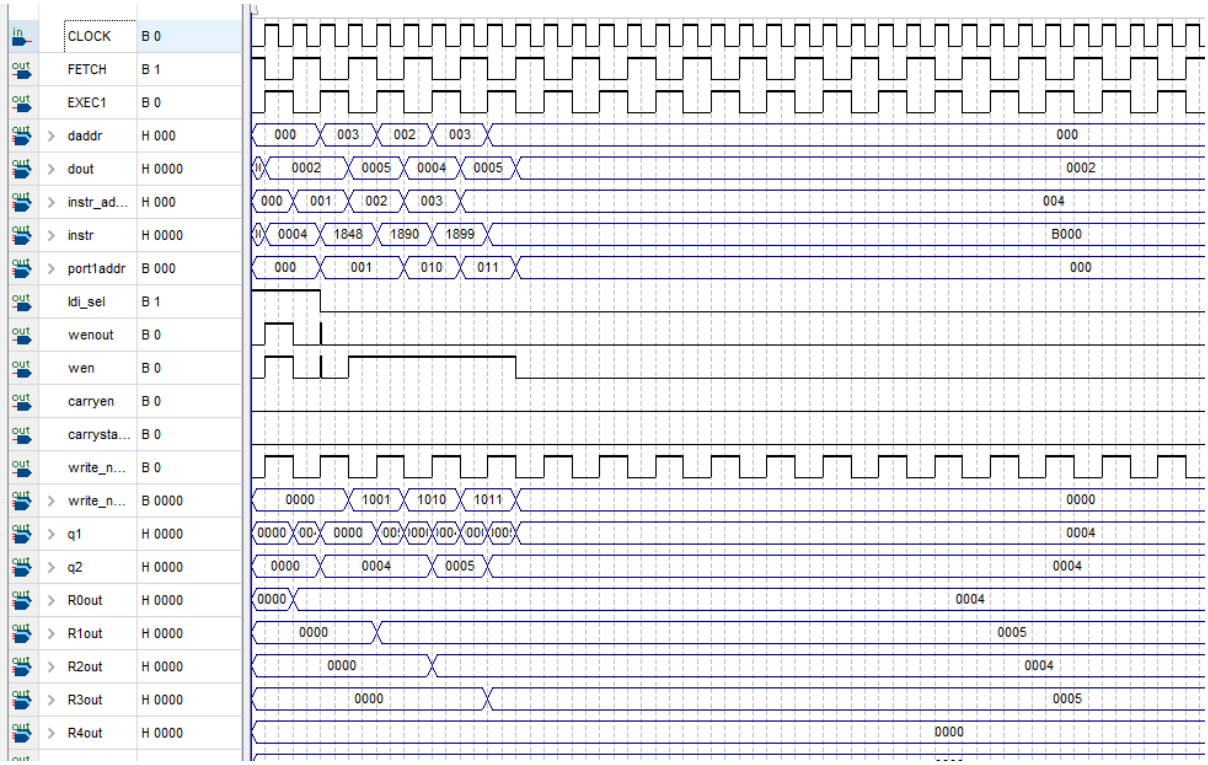
0x005

0x006

Instruction mem:

Instruction	**Machine code**	**Action**
LDI 0x4	0004	R0 = 0x0004
LDR 1 1 R1 R0	1848	R1 = mem[R0-1] = 0x0005
LDR 1 2 R2 R0	1890	R2 = mem[R0-2] = 0x0004
LDR 1 2 R3 R1	1899	R3 = mem[R1-2] = 0x0005
STP	B000	

Working:



- Can't write register after LDR, can only LDR, STR, JMP, STMFD and STP
  - What if must write register after LDR, how to wait?
    - Messy: Could have 2 separate registers for stack and link register to write at the same time - may be more messy
    - Hacky: have a JMP instruction to next PC value to basically do nothing or STR a value in memory
    - Best?: Have logic that says if when write_next_status[3] is on and the current instruction is a write instruction, PC stays at the current instruction
      - Can add input to decoder that stops pc_cnt_en just like stp
      - I believe the current order of MUXes will make LDR writing to register take priority over LDI for example

Implementation:
ALU:

```
// write_next_stp goes to decoder to stop PC
assign write_next_stp = write_next_status[3] & ~(ldr | str | jmp | stmfd | stp);
```

Decoder:

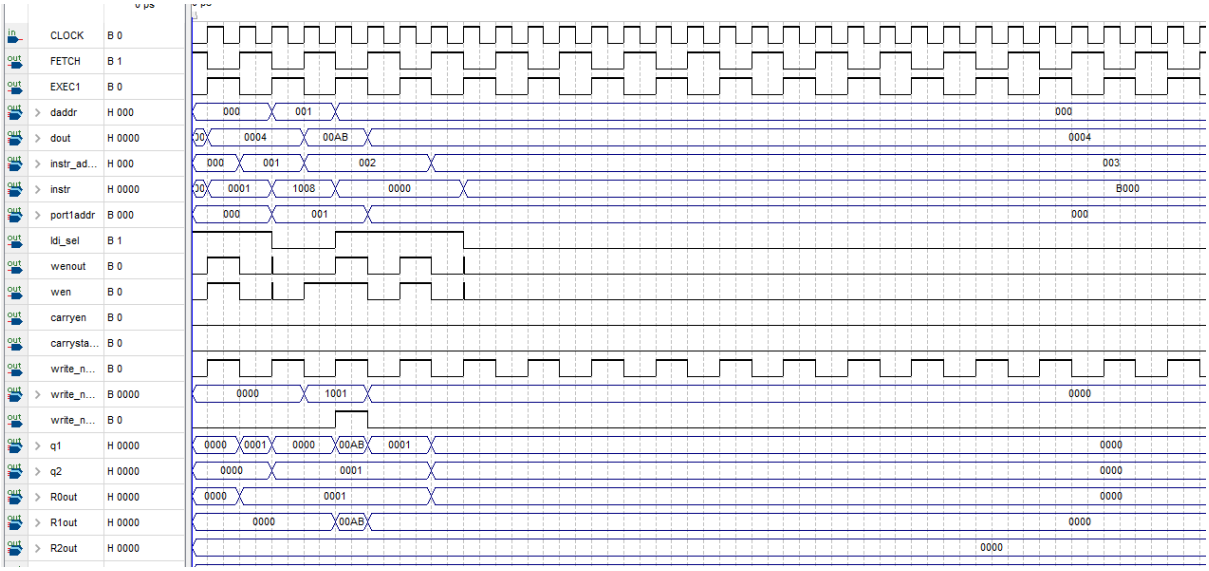
```
// Need to think of all decode signals and then implement
assign pc_sload = exec1 & jmp;
assign pc_cnt_en = exec1 & ~(jmp | stp | write_next_stp);
assign ram_wren = exec1 & (str | stmfd);
```

Test LDR and then LDI
Data memory:
0x0004
0x00AB

Instruction mem:

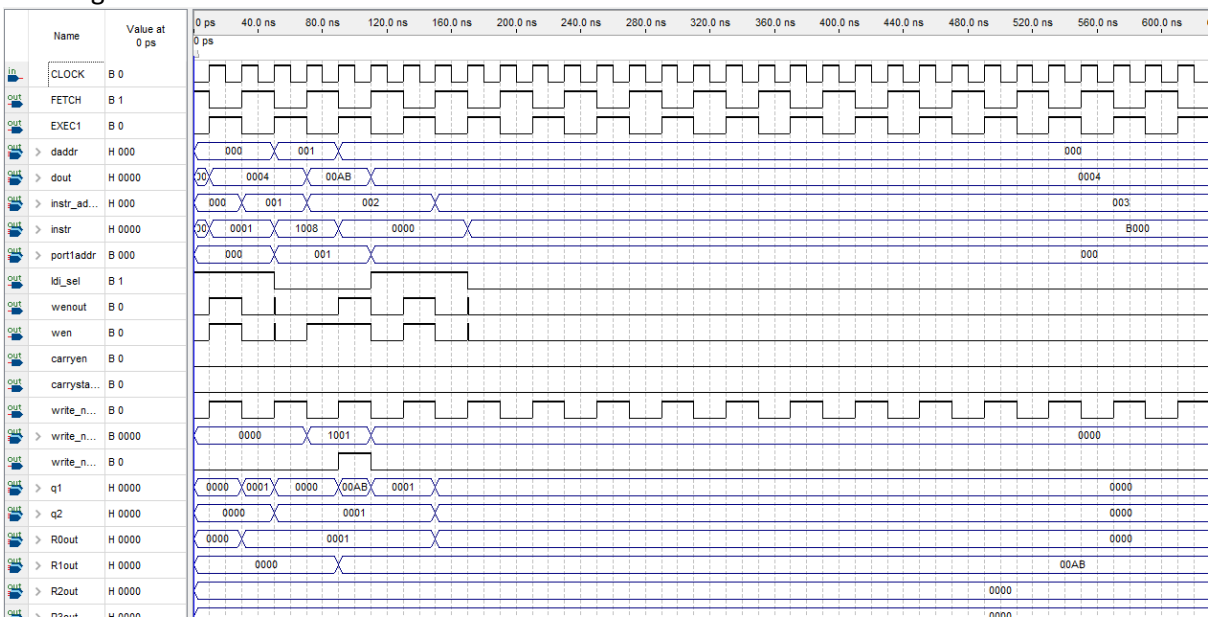
LDI 0x1	0001
LDR 0 0 R1 R0	1008
LDI 0x0	0000
STP	B000

Issue:



- R1 loads 0
  - When port1addr is still at 1, ldi has started already
  - should make sure when write_next_stp is on, ldi_sel is off
  - Also changed ldi_sel so it is in alu - makes more sense

Working:



Test Mov after LDR

Data memory:

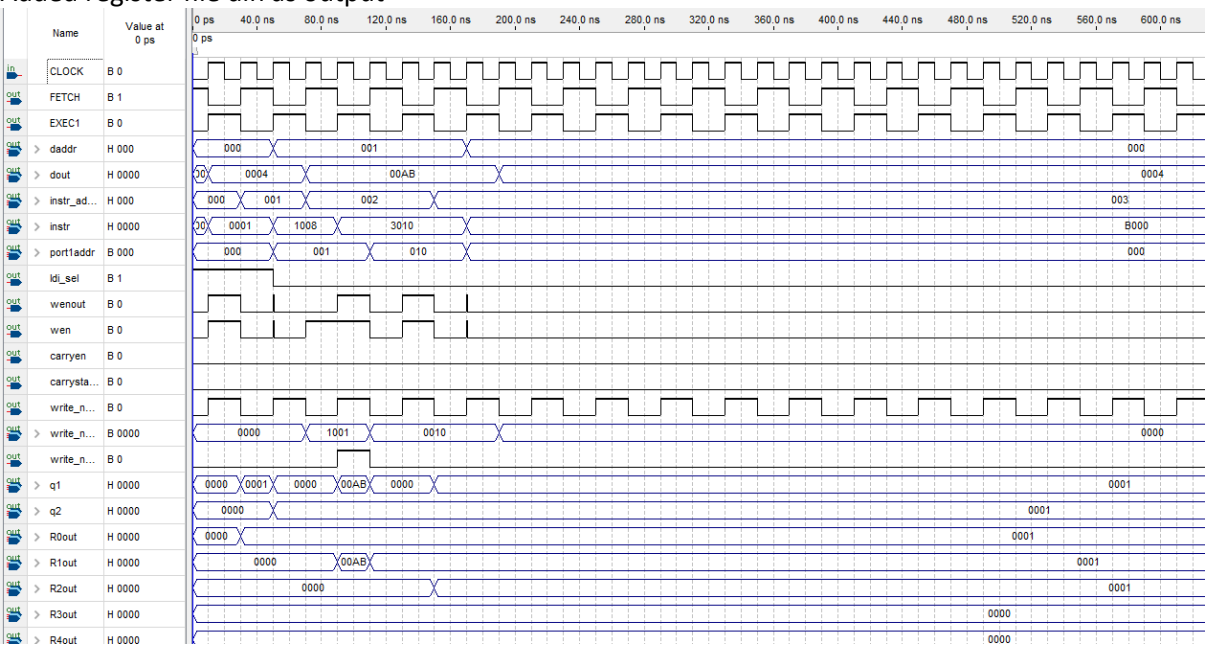
0x0004

0x00AB

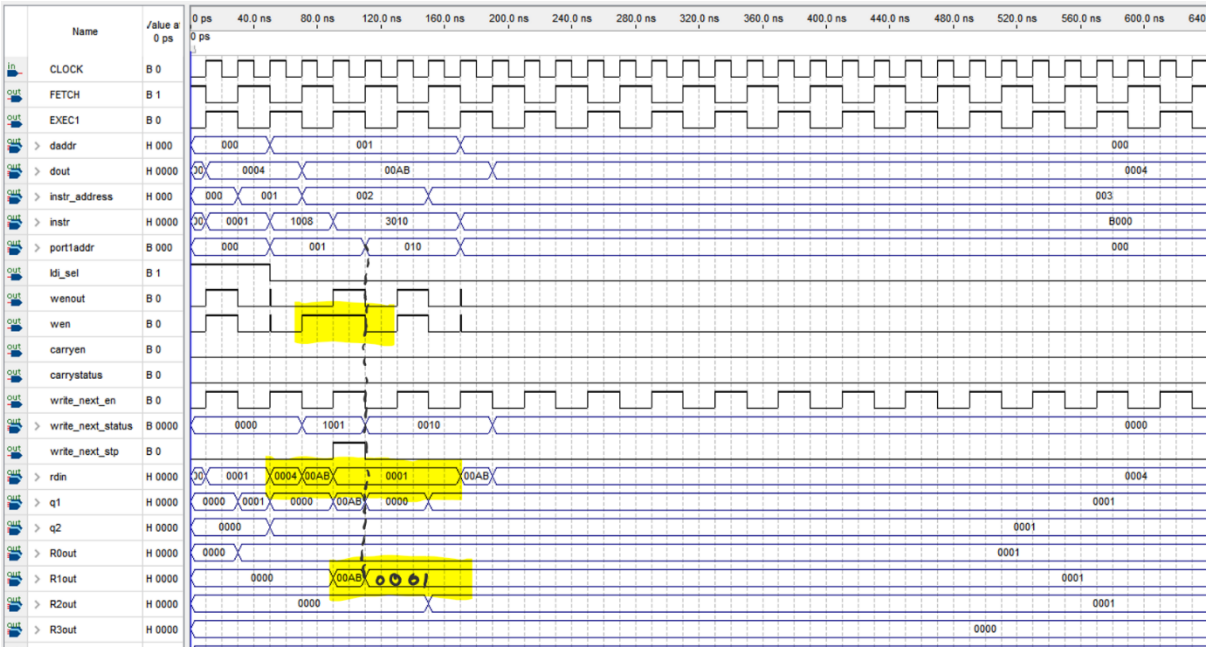
Instruction mem:

LDI 0x1	0001
LDR 0 0 R1 R0	1008
MOV 0 0 0 R2 R0	3010
STP	B000

- Added register file din as output



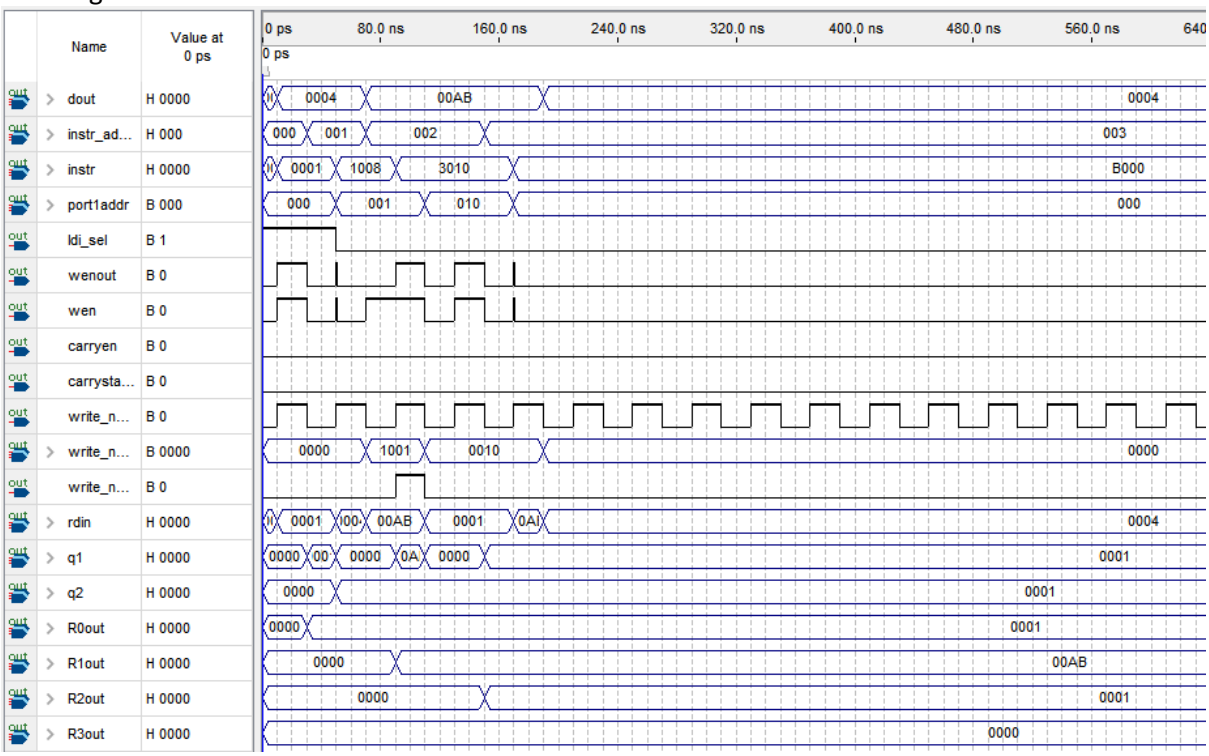
- Need a way to de-activate the current instruction from changing the
- Din should not be changed while port1addr is still at 1



- Din_sel should still select from din even though mov has started if the write_next_stp is enabled

```
// din_sel should be on if writing back to registers from alu (mov, add, sub, mul, bl)
// If ldr is writing, din should not be changed yet
assign din_sel = (mov | add | sub | mul | bl) & ~write_next_stp;
```

Working:



STR

STR (no offset) test:

Data mem:
0x0
0x2

Instruction mem:

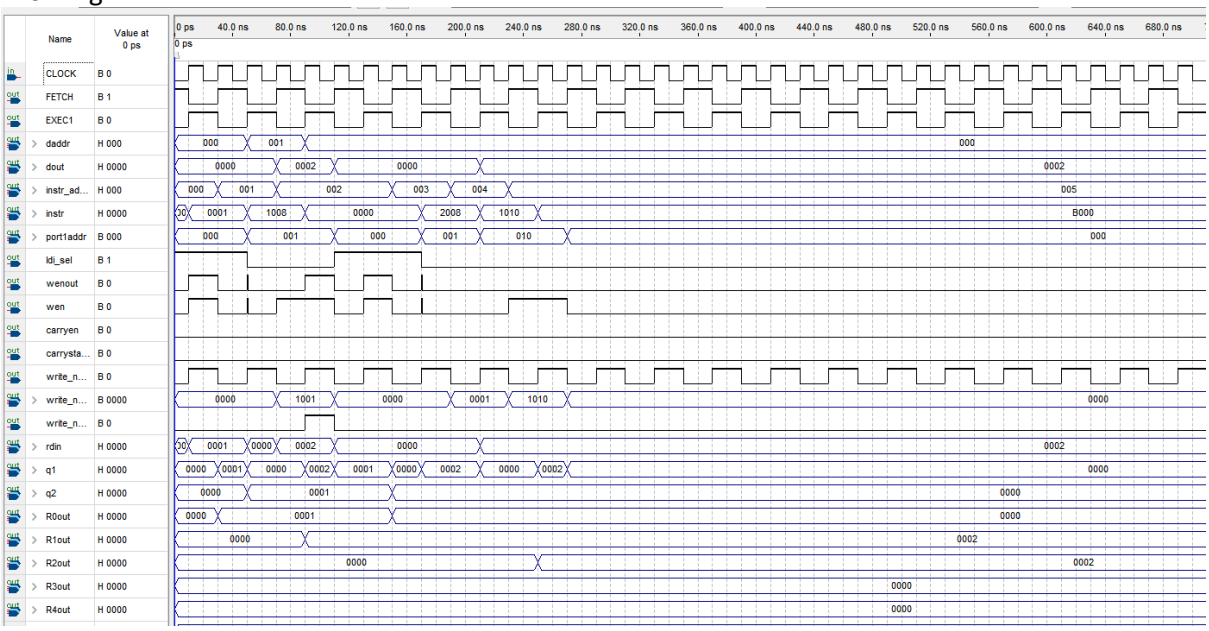
Instruction	Machine code	Action
LDI 0x1	0001	R0 = 0x1
LDR 0 0 R1 R0	1008	R1 = mem[R0] = 0x2
LDI 0x0	0000	R0 = 0x0
STR 0 0 R1 R0	2008	Mem[R0] = mem[0x0] = R1 = 0x2
LDR 0 0 R2 R0	1010	R2 = mem[R0] = 0x2
STP	B000	

- CANT LDI AFTER LDR

Issue

- Need to add STR to also change daddr

Working:



MOV

```
// carryen enables writing to carry register - writes when cwen is enabled for add, sub, and mov
assign carryen = exec1 & cwen & (add | sub | mov);
```

- Need to actually write carry ff during all mov instructions
  - "No shift" writes 0
  - LSL writes MSB
  - Shift right writes LSB

```
3'b0011 : begin
            always @(*) begin
                  case (instr[8:7])
                  2'b00 : alusum = {1'b0,rsdata} + cin; // MOV
                  2'b01 : alusum = {rsdata,cin} // LSL
                  2'b10 : alusum = {rsdata[0], cin, rsdata[15:1]};// Right shift - cin determines LSR or ASR
                  default : alusum = alusum = {1'b0,rsdata} + cin;
               endcase;
            end
         end
```

Test:
Data mem:
0xFFF0

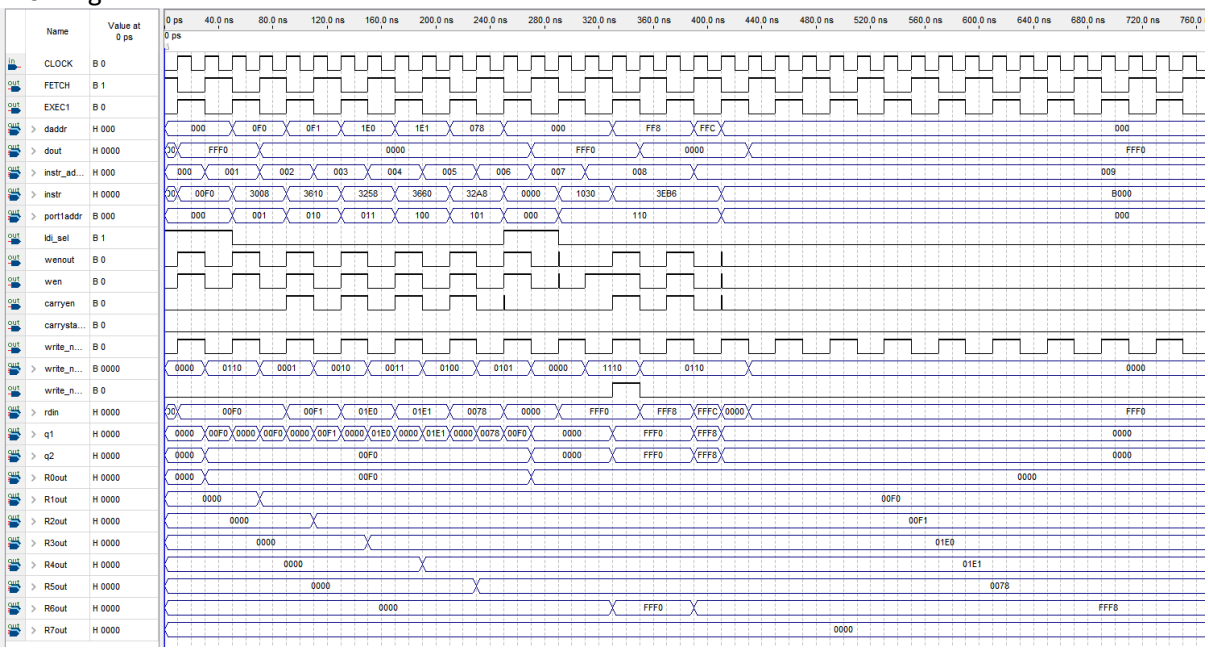
Instruction mem:

Instruction	Machine code	Action
LDI 0xF0	00F0	R0 = 0xF0
MOV 0 0 0 R1 R0	3008	R1 = R0 = 0x0F0
MOV 1 1 0 R2 R0	3610	R2 = R0 + 1 = 1 0xF1

MOV 0 1 1 R3 R0	3258	R3 = LSL R0 = 0x1E0
MOV 1 1 1 R4 R0	3660	R4 = 0x1E1
MOV 0 1 2 R5 R0	32A8	R5 = LSR R0 = 0x78
LDI 0	0000	R0=0
LDR 0 0 R6 R0	1030	R6 = 0xFFF0
MOV 3 1 2 R6 R6	3EB6	R6 = ASR 0xFFF0 = 0xFFF8
STP	B000	

00F0 3008 3610 3258 3660 32A8 0000 1030
3EB6 B000

Working:



JMP

- As the conditions for jump need to determine if sload is loaded, the decision was made to put the entire decoder in the ALU
  - This makes it easier and less messy, so multiple signals do not need to leave each block and result in longer time to compile and test

JMP ALU logic:

```
// conditional operators - compares Rd and comparator (in cond) - used for if jump should occur
wire eq = (rddata == cond); // Rd == comparator
wire mi = (rddata < cond); // Rd < comparator

reg jmp_cond; // change jmp cond to determine if jmp will occur depending on comparison and jump condition in instruction word

// PC and RAM control signals
assign pc_sload = exec1 & jmp_cond;
assign pc_cnt_en = exec1 & ~(jmp_cond | stp | write_next_stp);
assign ram_wren = exec1 & (str | stmfd);


// determine alusum - need to change opcodes
always @(*) begin
      case (op)
      4'b0001, 4'b0010 : alusum = sign ? (rsdata - {11'b0,offset}) : (rsdata + {11'b0,offset}); // LDR and STR: calculate daddr

      4'b0100 : begin
                  case (field)
                  2'b00 : jmp_cond = 1; // JMP: Always jump
                  2'b01 : jmp_cond = eq; // JEQ: Jump if Rd == comparator
                  2'b10 : jmp_cond = mi; // JMI: Jump if Rd < comparator
                  2'b11 : jmp_cond = ~eq & ~mi; //JMB: Jump if Rd > comparator
                endcase;
            end
```

- Did not work - pc_sload and pc_cnt_en were undefined during exec1

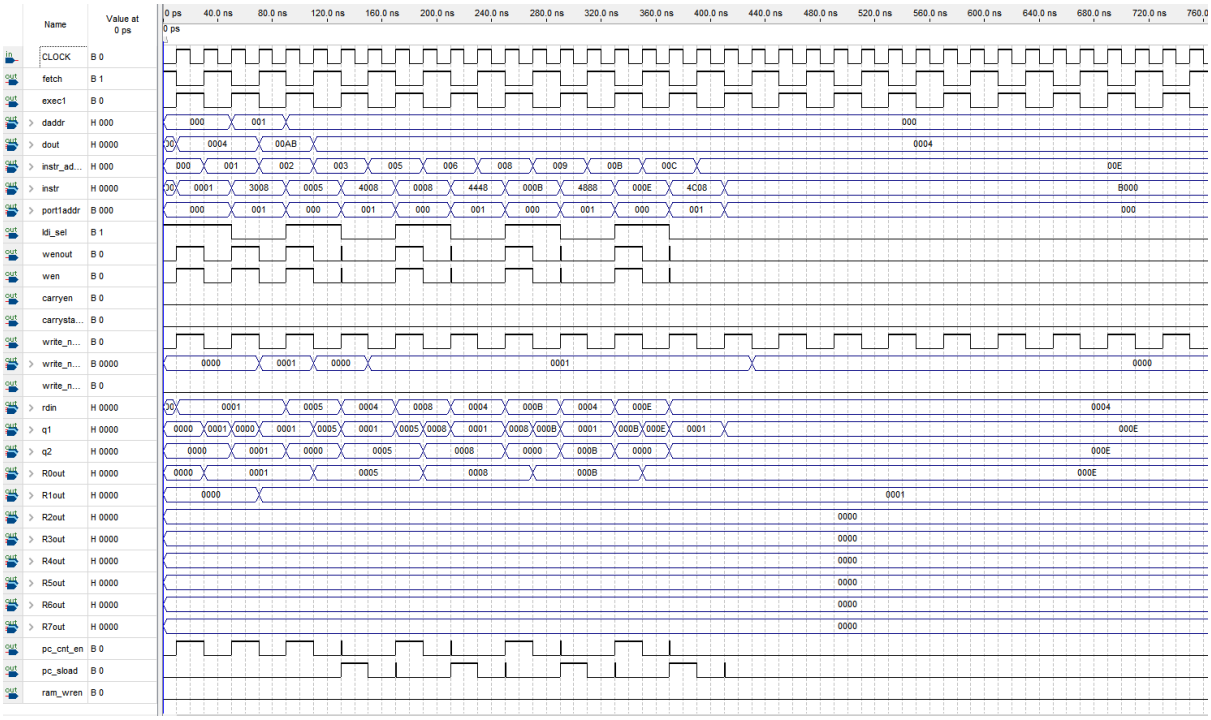
JMP test

Instruction mem:

PC	Instruction	Machine code	Action
0	LDI 0x1	0001	R0=0x1
1	MOV 0 0 0 R1 R0	3008	R1 = R0 = 0x1
2	LDI 0x5	0005	R0 = 0x5
3	JMP 0 0 R1 R0	4008	JMP 5 // set PC=5
4	LDI 0xFFF	0FFF	R0=0xFFF // JMP not working
5	LDI 0x8	0008	R0 = 0x8
6	JMP 1 1 R1 R0	4448	JEQ 8 // set PC=8
7	LDI 0xFFF	0FFF	R0=0xFFF // JEQ not working
8	LDI 0xB	000B	R0=0xB
9	JMP 2 2 R1 R0	4888	JMI 11 // Set PC=11
10	LDI 0xFFF	0FFF	R0=0xFFF // JMI not working
11	LDI 0xE	000E	R0=0xE
12	JMP 3 0 R1 R0	4C08	JMB 14 // Set PC=14
13	LDI 0xFFF	0FFF	R0 = 0xFFF // JMB not working
14	STP	B000	

0001 3008 0005 4008 0FFF 0008 4448 0FFF
000B 4888 0FFF 000E 4C08 0FFF B000

Working:



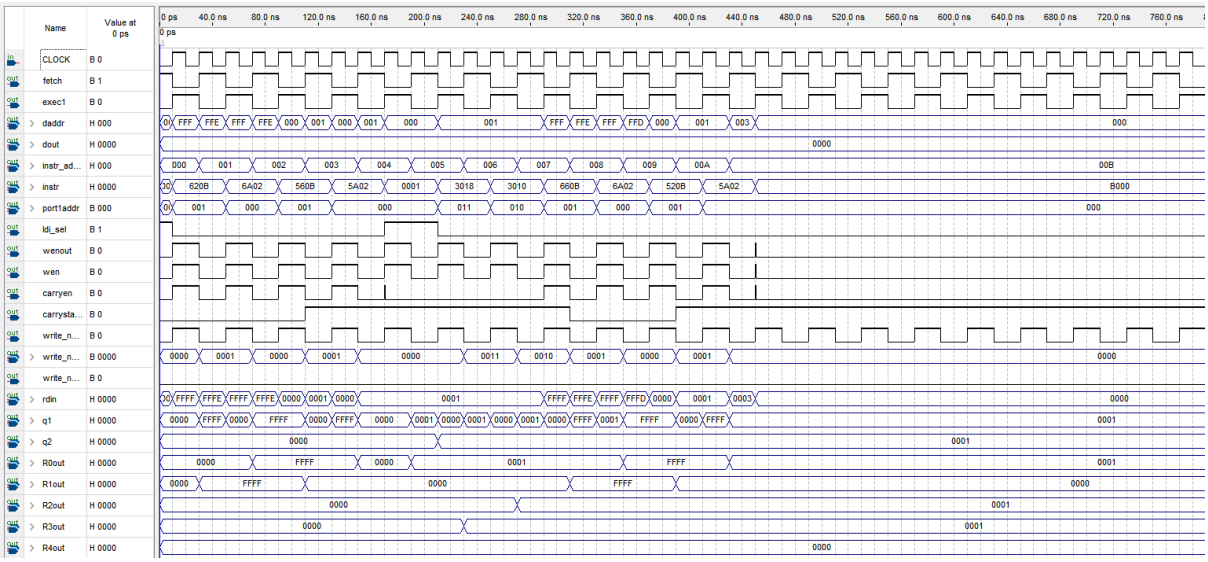
ADD/SUB

Multi-register Add/Sub test (from ARMish testing)

Instruction mem:

Instruction	Machine code	Action
SUB 0 1 R1 R3	620B	
SUB 2 1 R0 R2	6A02	R0:R1 := R0:R1 - R2:R3 - 1
ADD 1 1 R1 R3	560B	
ADD 2 1 R0 R2	5A02	R0:R1 := R0:R1 + R2:R3 + 1
LDI 0x1	0001	R0=0x1
MOV 0 0 R3 R0	3018	R3=0x1
MOV 0 0 R2 R0	3010	R2=0x1
SUB 1 1 R1 R3	660B	
SUB 2 1 R0 R2	6A02	R0:R1 := R0:R1 - R2:R3
ADD 0 1 R1 R3	520B	
ADD 2 1 R0 R2	5A02	R0:R1 := R0:R1 + R2:R3
STP	B000	

Working:

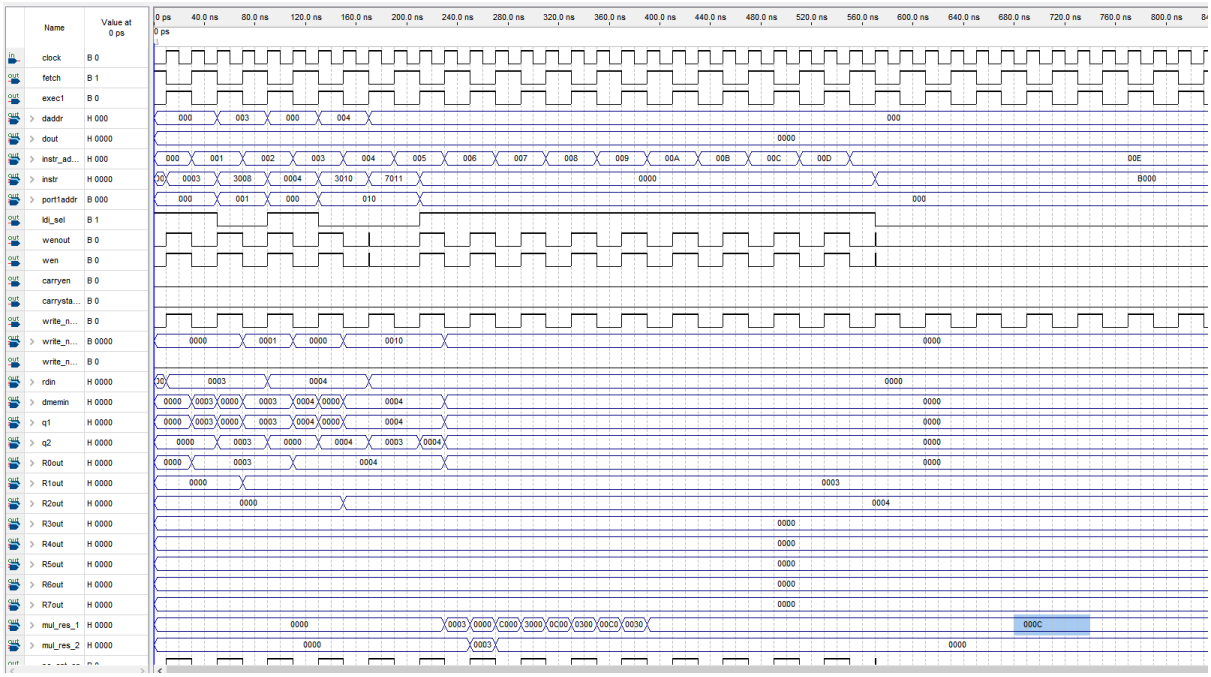


MUL

Instruction	Machine code	Action
LDI 0x3	0003	
MOV R1 R0	3008	
LDI 0x4	0004	
MOV R2 R0	3010	
MUL R2 R1	7011	
LDI 0x0	0000	
LDI 0x0		
LDI 0x0		
LDI 0x0		
LDI 0x0		
LDI 0x0		
LDI 0x0		
LDI 0x0		
LDI 0x0		
LDI 0x0		
STP	B000	

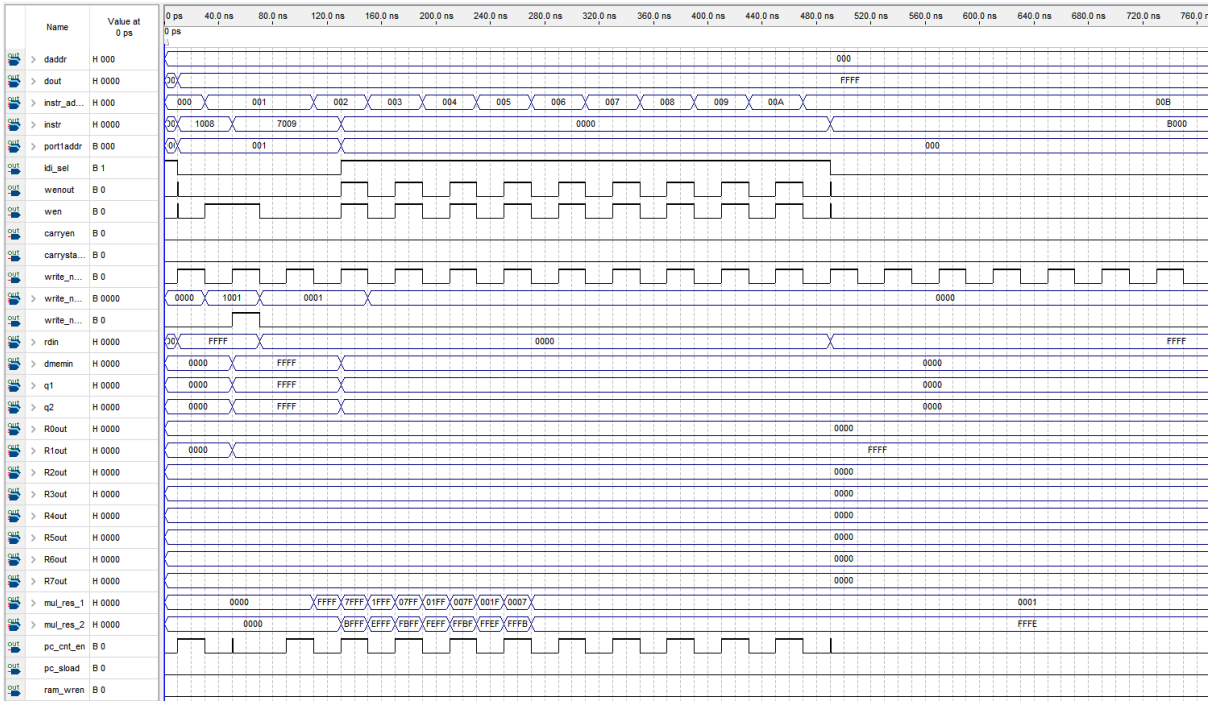
- Need to make sure regfile is not written during MUL

  Working:



Largest Multiplication:
Data RAM:
0xFFFF

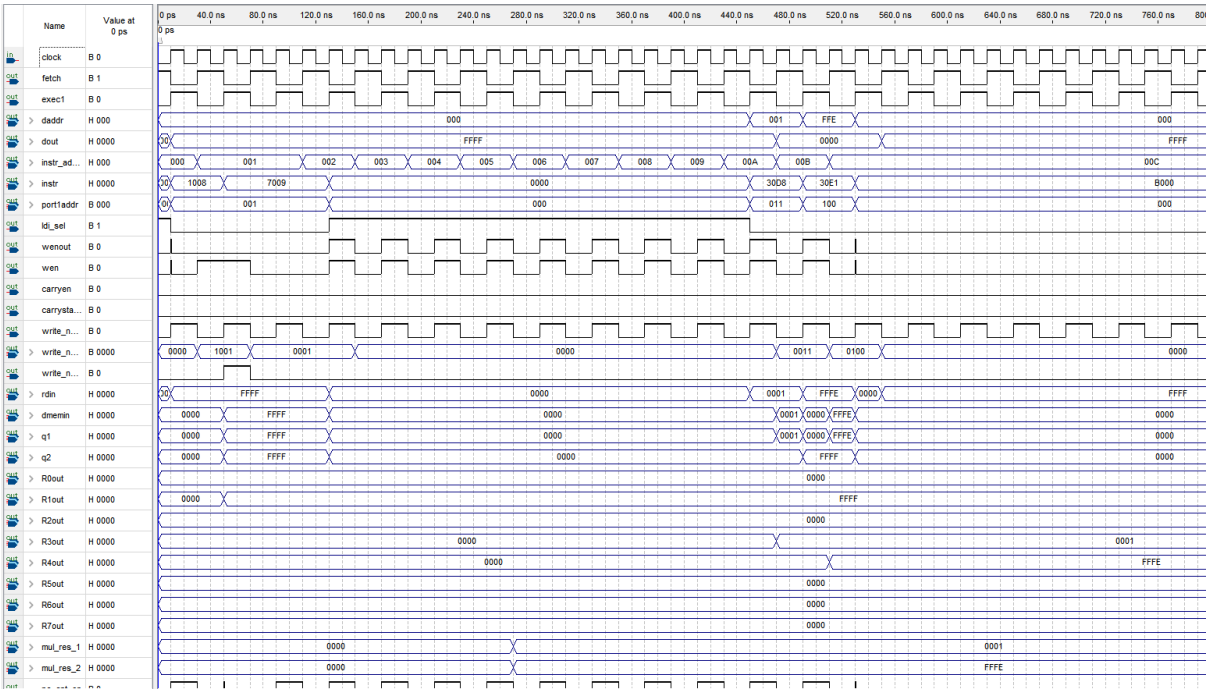
Instruction	Machine code	Action
LDR 0 0 R1 R0	1008	
MUL R1 R1	7009	
LDI 0x0	0000	
LDI 0x0		
LDI 0x0		
LDI 0x0		
LDI 0x0		
LDI 0x0		
LDI 0x0		
LDI 0x0		
LDI 0x0		
LDI 0x0		
STP	B000	



Storing MSB and LSB into regfile

Instruction	Machine code	Action
LDR 0 0 R1 R0	1008	
MUL R1 R1	7009	R1 * R1
LDI 0x0	0000	
LDI 0x0		
LDI 0x0		
LDI 0x0		
LDI 0x0		
LDI 0x0		
LDI 0x0		
LDI 0x0		
MOV 0 0 3 R3 0	30D8	R3 = LSBs of product
MOV 0 0 3 R4 1	30E1	R4 = MSBs of product
STP	B000	

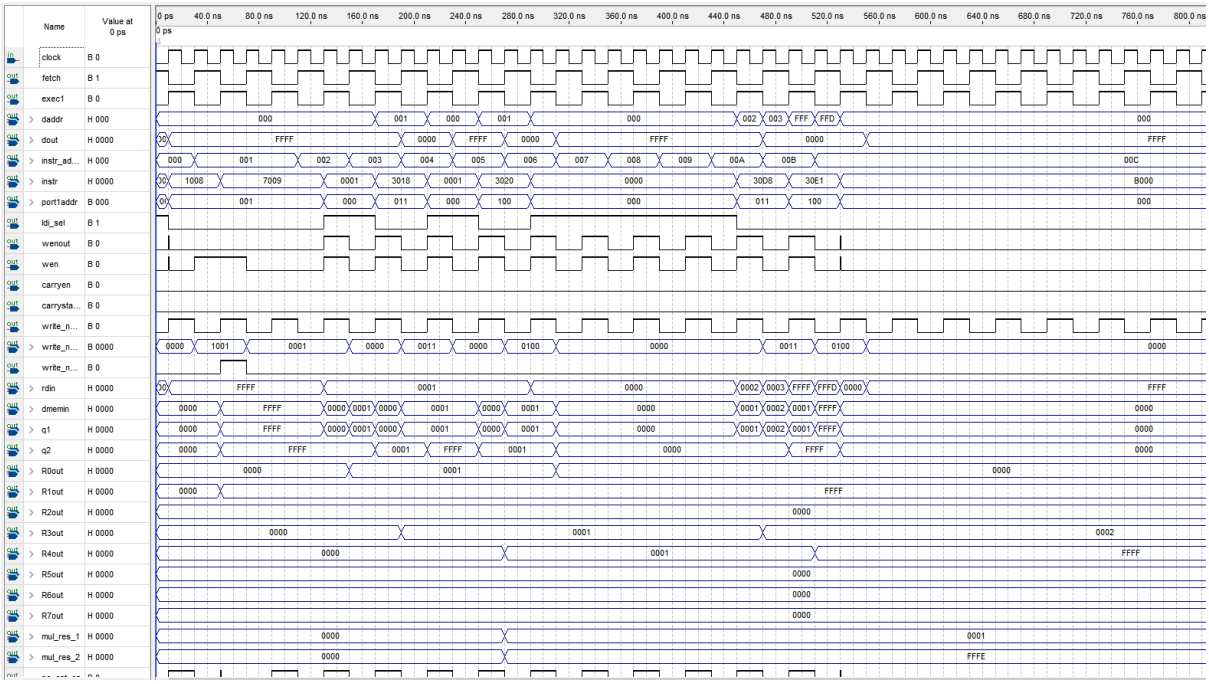
1008 7009 0000 0000 0000 0000 0000 0000
0000 0000 30D8 30E1 B000



Testing add and multiply:

Instruction	**Machine code**	**Action**
LDR 0 0 R1 R0	1008	
MUL R1 R1	7009	R1 * R1
LDI 0x1	0001	
MOV R3 R0	3018	
LDI 0x1	0001	
MOV R4 R0	3020	
LDI 0x0	0000	
LDI 0x0		
LDI 0x0		
LDI 0x0		
MOV 0 0 3 R3 0	30D8	R3 = LSBs of product+1
MOV 0 0 3 R4 1	30E1	R4 = MSBs of product+2
STP	B000	

1008 7009 0001 3018 0001 3020 0000 0000
0000 0000 30D8 30E1 B000



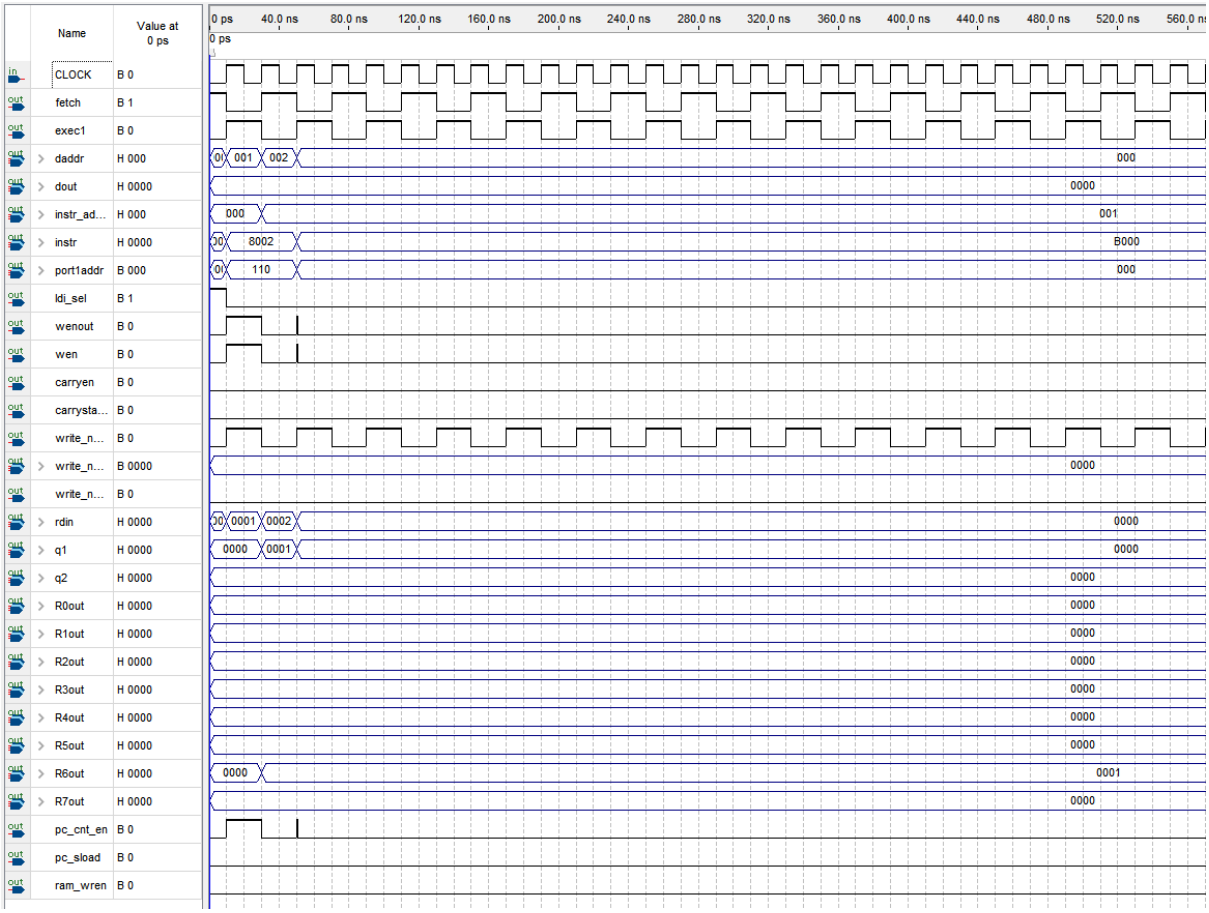
BL

Test if link register loading works

Instruction mem:

PC	Instruction	Machine code	Action
0	BL 2	8002	For now just sets R6=0x1
1	STP	B000	

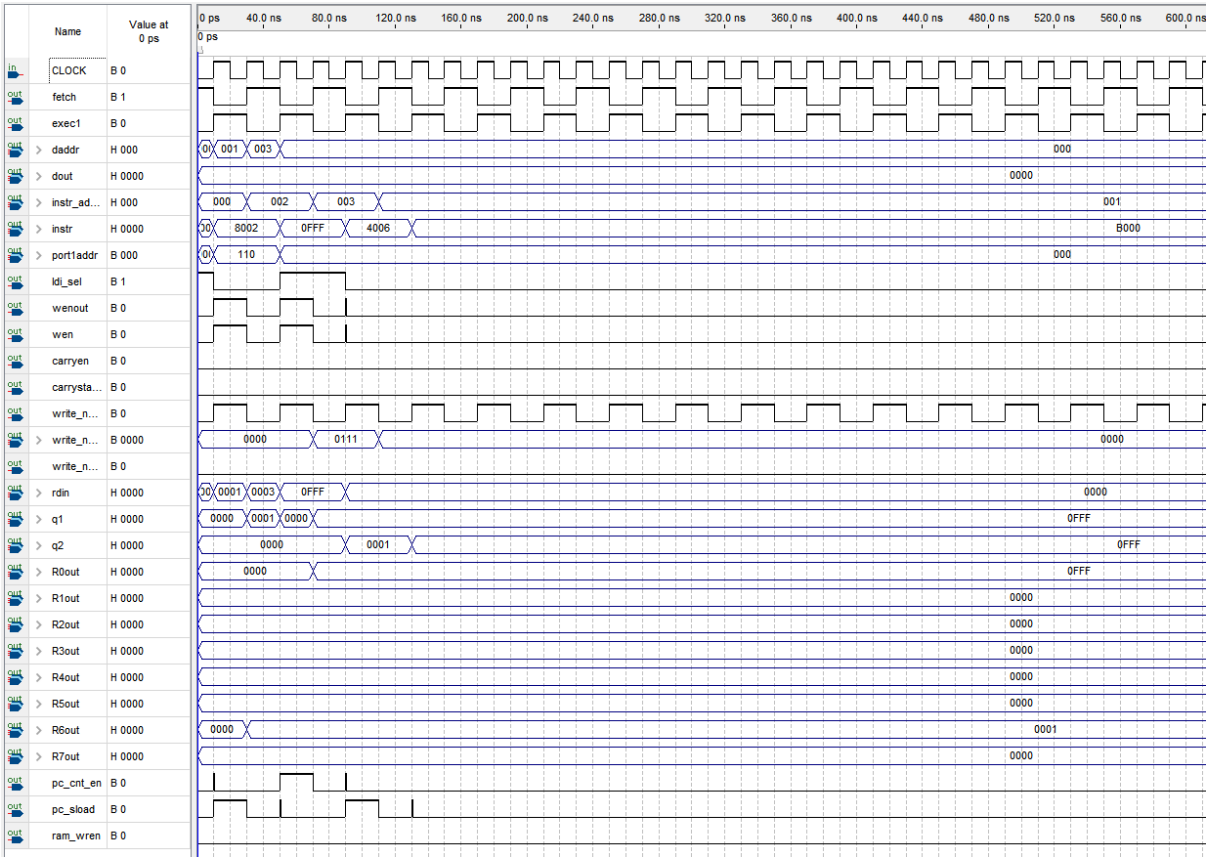
Working:



Test if full BL works

PC	Instruction	Machine code	Action
0	BL 2	8002	For now just sets R6=0x1
1	STP	B000	
2	LDI 0xFFF	0FFF	R0=0xFFF
3	JMP 0 0 R0 R6	4006	PC=0x1

Working:



STACK

LDMFD and STMFD

Test:

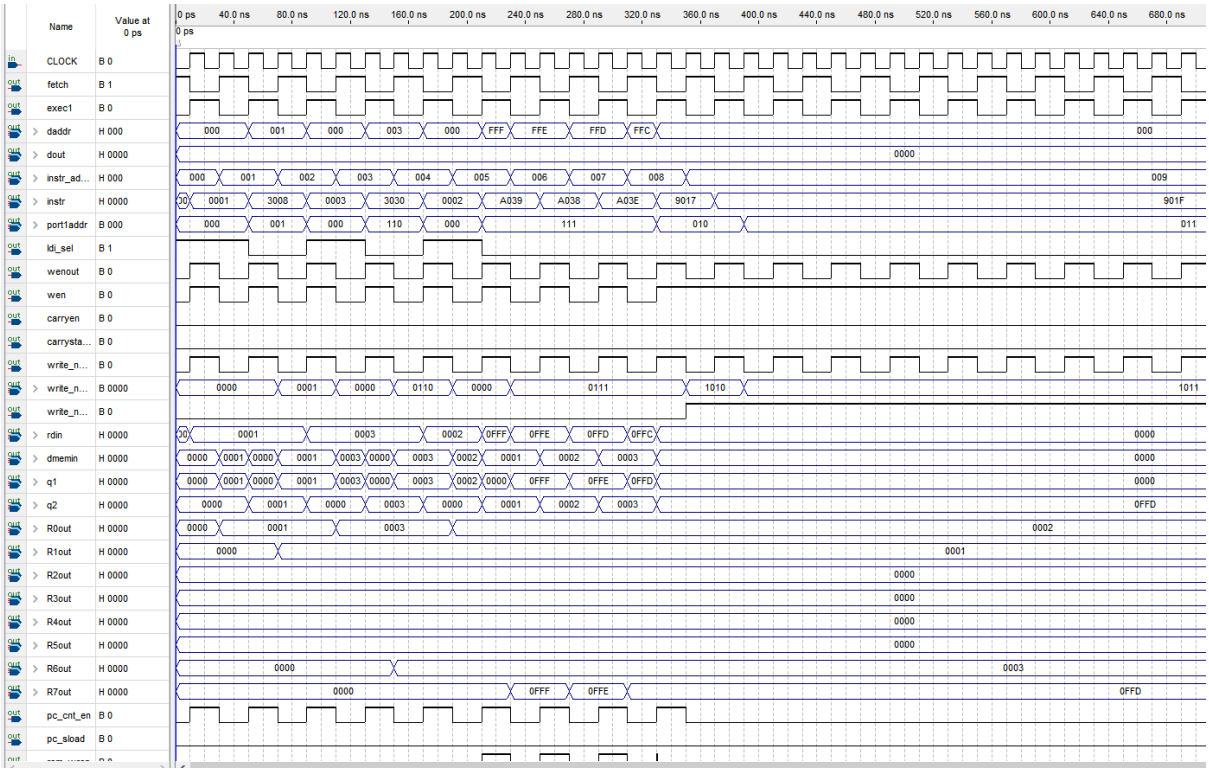
Instruction	Machine code	Action
LDI 0x1	0001	
MOV R1 R0	3008	R1=0x1
LDI 0x3	0003	
MOV R6 R0	3030	R6=0x3
LDI 0x2	0002	R0=0x2
PUSH R1	A00F / A039	PUSH 0x1
PUSH R0	A007 / A038	PUSH 0x2
PUSH R6	A037 / A03E	PUSH 0x3
POP R2	9017	R2=0x3
POP R3	901F	R3=0x2
POP R4	9027	R4=0x1
STP	B000	

0001 3008 0003 3030 0002 A039 A038 A03E
9017 901F 9027 B000

Issue:

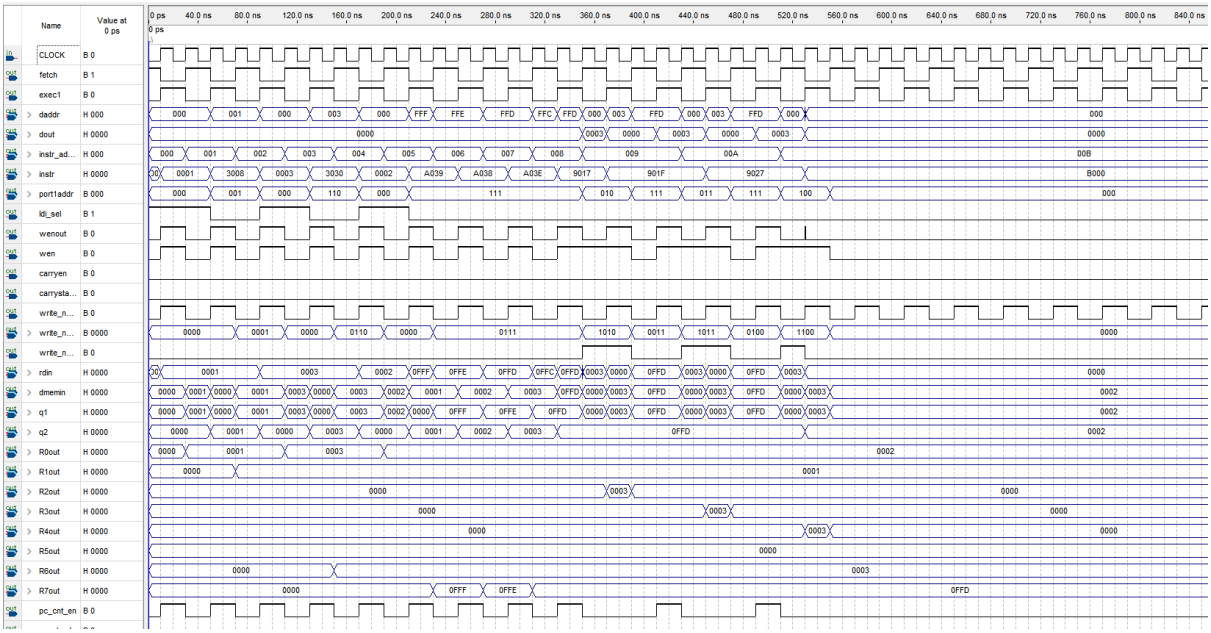
- Stack pointer is not changing
    - Rdin is rd-1
    - Also need to have wen during stmfd
- We also need aluout to be selected for din
- Thus, ldmfd instructions cannot be done in parallel like ldr as something is written every cycle

- Now it is writing R6, instead of LDMFD and STMFD
- Rdin should be



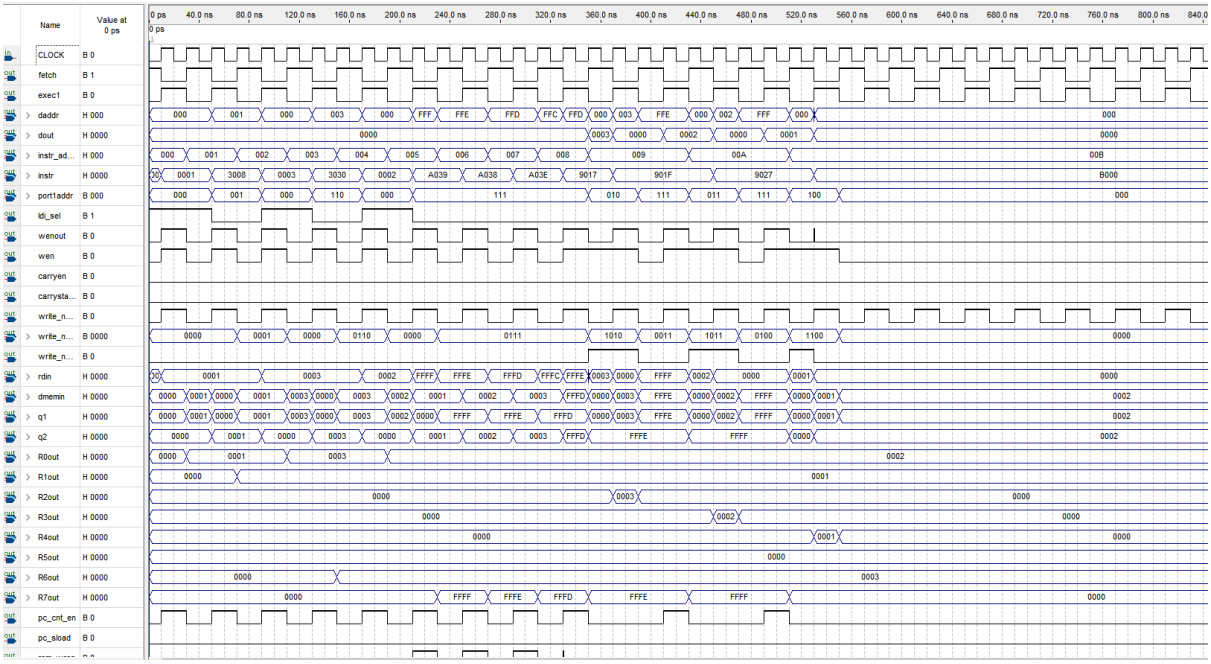
- Stmfd works but not ldmfd
- 2 ldmfd instructions do not work after each other as write_next_stp is stuck at 1

Issue:



- STMFD works but in LDMFD, stack pointer is not updating

Issue:

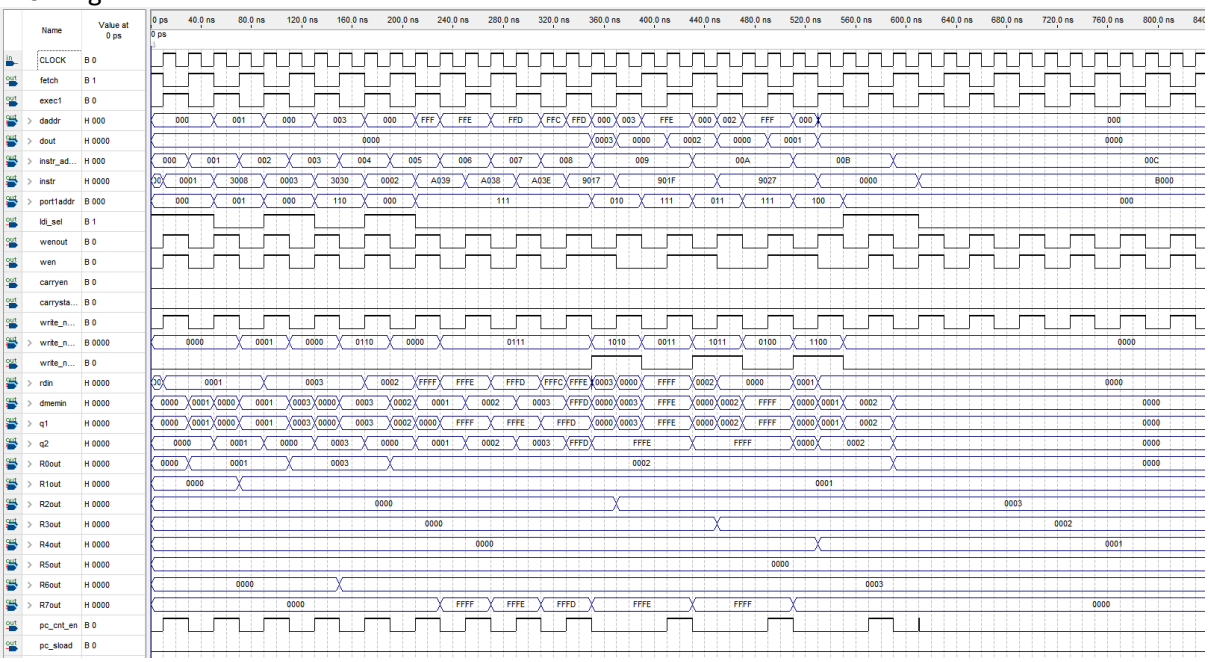


- Works but the registers are rewritten to 0 as rdin becomes 0
- This is because dout changes to 0 for the instructions that need to wait when write_next_stp is enabled
    - Daddr switches to 0 during fetch as aluout is reset

- Saw that when write_next_flag is on, wen should be the inverse of wenout, so that it is not on in that last cycle where 0 is written is at daddr, so could use an XOR gate to carry out selective inversion
    - May cause problems when pipelining

- Added stp to wenout, this may cause problems for other instructions writing during stp - check later

Working:

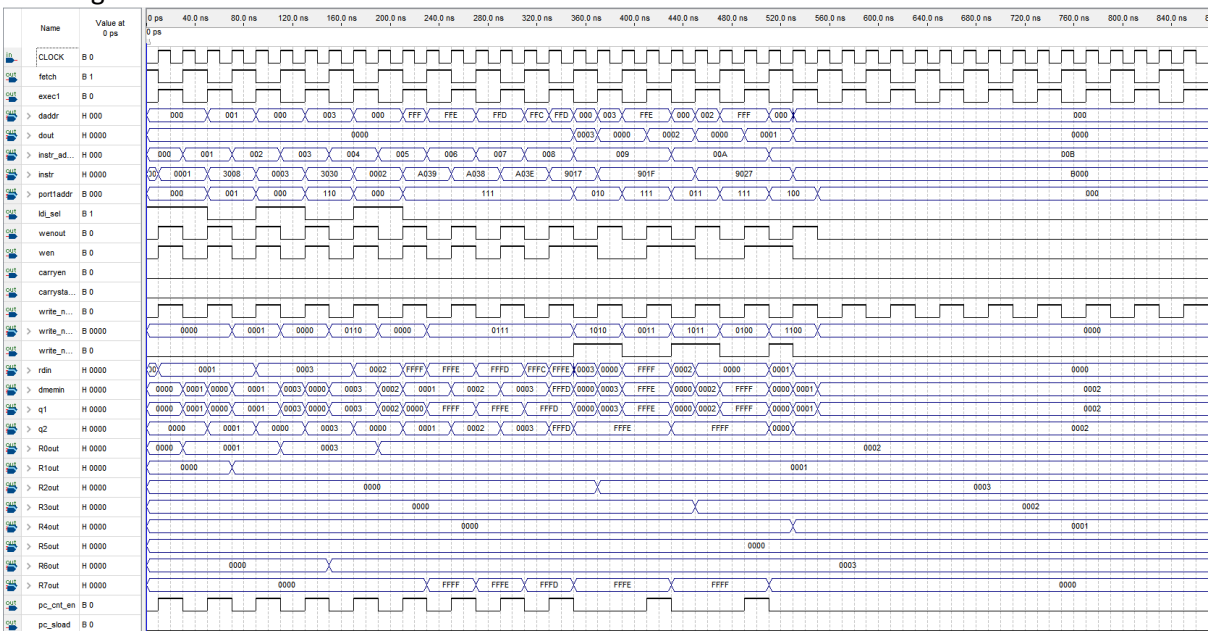


Issue: loading 0 in because enabled stp in wrenout

```
assign wenout = exec1 & (ldi | mov | add | sub | mul | bl | ldmfd | stmfd | stp & write_next_status[3]);
```

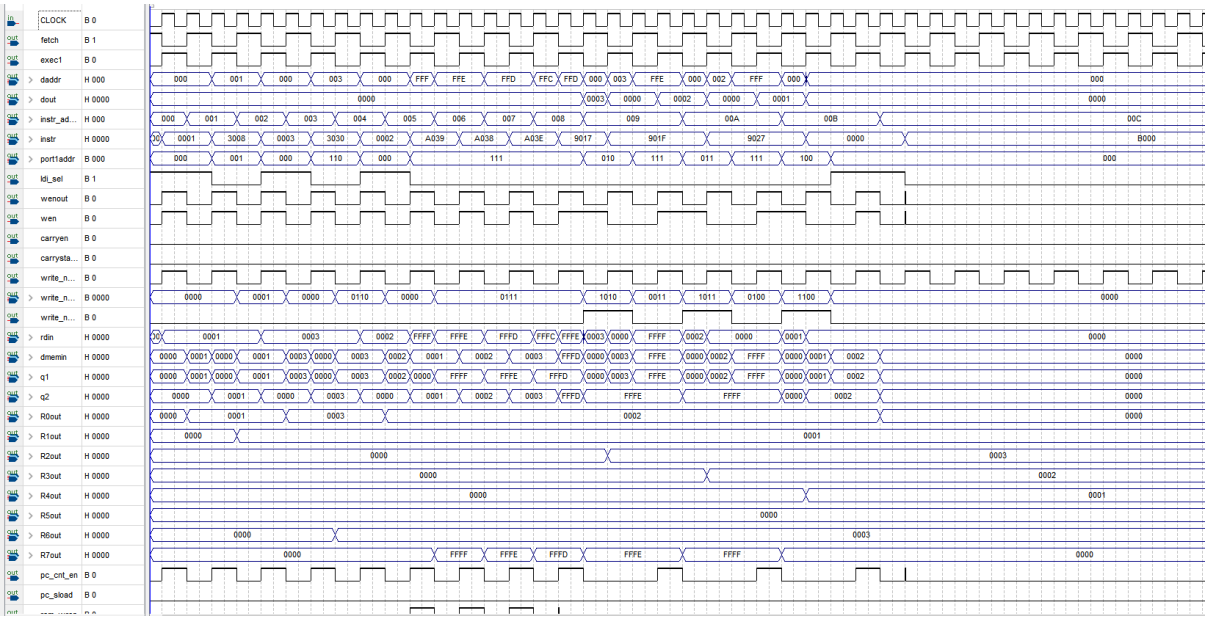
Added this - may help - temporary solution - may be different when pipelining

Working:



Test doing LDI after this to make sure it works after POP

- Works with LDI


CLOCK B 0
fetch B 1
exec1 B 0
daddr H 000
dout H 0000
instr_ad... H 000
instr H 0000
port1addr B 000
ldi_sel B 1
wenout B 0
wen B 0
carryen B 0
carrysta... B 0
write_n... B 0
write_n... B 0
write_n... B 0000
rdin H 0000
dmemin H 0000
q1 H 0000
q2 H 0000
R0out H 0000
R1out H 0000
R2out H 0000
R3out H 0000
R4out H 0000
R5out H 0000
R6out H 0000
R7out H 0000
pc_cnt_en B 0
pc_sload B 0