Harvard Architecture 16-Bit CPU Design and Synthesis (December 2021)

Long Nguyen, Jonathan Case, *Student*, *IEEE*.

Abstract—This paper documents the design, implementation, simulation, synthesis and operation of a 16-bit central processing unit in Verilog. This paper showcases the thought process behind the choices made in the design and implementation such as the bit field makeup of the 16-bit instructions. This paper discusses how simulation assisted with verifying the theoretical design and implementation, and synthesis on the DE1-SoC provided a verification of the hardware used in the design. This paper outlines the benefits and challenges of bit limitations in the operation of the unit.

Index Terms—ALU, Computer Architecture, CPU, Datapath, DE1-SoC, FPGA, Memory, ModelSim, Pipeline, Quartus Prime, Synthesis, Verilog.

I. Introduction and Background

MOST modern computers operate on an optimized 32 or 64 bit architecture. This allows for many complex instructions and operations to be utilized. However, with fewer bits in a CPU instruction, there are less options for the number of unique operations, accessible registers, and immediate fields. This restriction was the inspiration for this project. Our plan was to design, implement, and verify a CPU with a reduced instruction set.

II. Goals and Specifications

The goal of this project is to create a 16-bit Harvard CPU that includes our custom Reduced Instruction Set Computer architecture.

0.	0000	add	rd, rs, rt
1.	0001	addi	rd, rs, imm
2.	0010	sub	rd, rs, rt
3.	0011	and	rd, rs, rt
4.	0100	or	rd, rs, rt
5.	0101	xor	rd, rs, rt
6.	0110	not	rd, rs
7.	0111	slt	rd, rs, rt
8.	1000	lsl	rd, rs, imm
9.	1001	lsr	rd, rs, imm
10.	1010	ldr	rt, rs, offset
11.	1011	str	rt, rs, offset
12.	1100	b	label(12)
13.	1101	bl	label(8)
14.	1110	br	rs
15.	1111	beq	rt, rs, label(4)

Fig. 1. Instruction set list. The parenthesis on some instructions indicate the bit length of the immediate field.

- General Purpose Registers: r0 to r12
- Input Register: r12
- Output Display Register: r0
- Special Purpose Registers:
  - (r13) zero register (immutable)
  - (r14) stack pointer
  - (r15) link register

Fig. 2. Registers file. These are built as temporary storage for the programmer to interact with the CPU.

Fig. 1 shows the comprehensive list of all supported instructions of our design. The first column indicates the opcode for each instruction, second column for the instruction name, and third column for the source/destination/transfer registers, immediate values, or a branch label value. A c++ assembler was created to parse a string assembly .txt file into the correct machine code binary output.

III. Design and Implementation

A. *General Architecture Design*

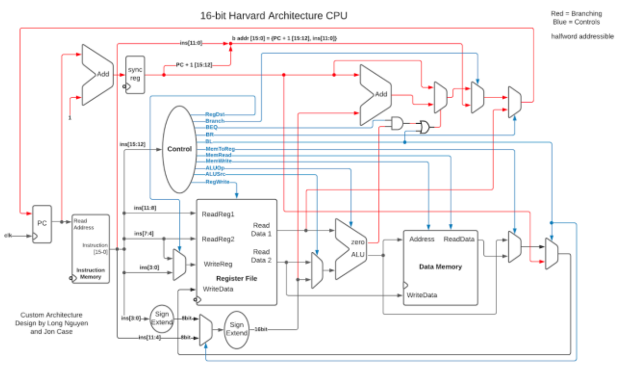


Fig. 3. Architecture for the 16-bit CPU with similarities to MIPS. Blue signals indicate control signals, while red indicates the various branch paths.

Our design was initiated by referencing a 32-bit Harvard CPU architecture similar to MIPS while the instruction set was

This manuscript was submitted on December 2021 as part of the ECE 174 Final Design Project for Dr. Razouk at CSU Fresno. All design files, including block diagram schematics, assembler, and verilog, were submitted accordingly. They can also be viewed on the following github repository: https://github.com/lhn1703/cpu_16bit.

inspired by ARM. This provided the base layout and port connections of the program counter, instruction memory, register file, data memory, and all combinational logic components. The first modification to the architecture was the instruction bit fields. We restructured the instruction bit fields so that the overall instruction bit-length is 16 bits. The opcode, source register, transfer register, destination register, and immediate fields were all set to 4 bits. This meets our design specification of 16 registers in the register file and our 16 unique instructions. However, this left only 4 bits to represent the immediate fields. For immediate signed values, this would only allow representations of -8 to 7. This design was also made such that the program counter, instruction memory, and data memory were halfword addressable. The decision to not make the memory blocks byte addressable was due to the fact that none of the instructions in our design made use of individual bytes and there was not any room to add more types of instructions in the future since we were already making use of every opcode combination.

B. *Considerations for FPGA Synthesis*

The design target was for the DE1-SoC FPGA board, which did not feature block RAMs asynchronous reads. Thus, the problem was circumvented by offsetting the instruction and data memory to different clock edges. In order to synchronize the data, the instruction memory performs a read at the negative edge while the data memory read and write are performed at the positive edge. Since the CPU will be physically synthesized on the development board, it was determined to be too tedious for the user to enter in one instruction at a time through the I/O switches and buttons on the board. Instead, the instruction memory will be a ROM that loads a pre-compiled assembly program. Thus this allows the user to run any program that requires an input (directly loaded into r12 from the switches) and returns a hexadecimal output to the 7-segment displays (the contents of r0) after pressing the reset button. Since there were only 10 switches, we decided to limit the input to be a 10-bit 2s complement number with sign extension from bit 9 to 15. In order to run different programs, the user must load them into the instruction memory, however they can execute the preloaded program for any number of times with any (reasonable) inputs.

C. *ALU Design*

The arithmetic logic unit is the heart of any CPU, performing all the key operations. One reliable and efficient type of adder is the carry-lookahead adder. We chose this type of adder for the ALU, because we wanted to prioritize performance over area. An additional benefit is that it is easy to implement via verilog generate loops.

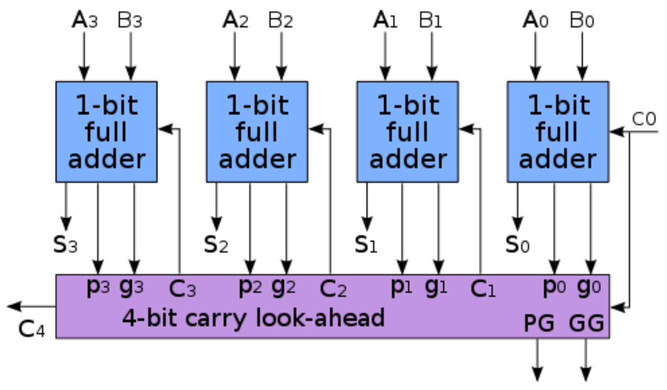


Fig. 4. Diagram for a 4-bit carry-lookahead adder.

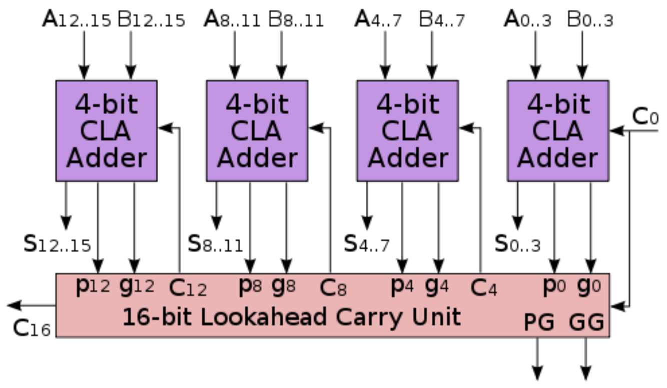


Fig. 5. Diagram for a 16-bit carry-lookahead adder.

Fig. 4. and fig. 5 shows the hierarchy of our 16-bit adder. The adder was reused for the addi, sub, beq, slt, ldr, and str instructions. Depending on the ALU opcode, the adder was configured to perform subtraction by inverting the second operand and setting the carry-in input to a logic high.

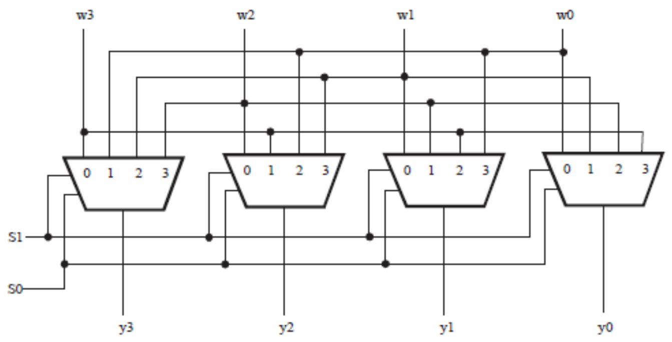


Fig. 6. Diagram for a 4-bit barrel shifter. The 16-bit version was implemented behaviorally via case statements.

For the lsl and lsr instructions, a barrel shifter design was implemented. This particular design was chosen once again due to its performance. All other logic instructions, such as bitwise AND and XOR were built from their logic gate equivalents.

D. *Memory Storage Design*

The next important structure of this CPU is the three blocks containing program information: instructions, registers, and data. The original MIPS implementation had asynchronous reads for the instructions and data block, however as mentioned previously this was adapted to utilize the block RAMs on the

DE1 FPGA. However, the registers file was constructed as asynchronous latches since it only stores 16 halfwords for quick access. For this design, the instruction and data memory contains 1024 halfwords (can be modified in the macro_defines.v file). The stack pointer always gets initialized to 768 whenever the CPU resets, which allocates the top quarter of the data memory to be used for the stack. The stack pointer points to the next free space on the stack, which means the stack can be categorized as empty ascending. Note, the programmer must manually allocate space on the stack for in their program by incrementing and storing to sp with appropriate offsets.

E. Building the Datapath

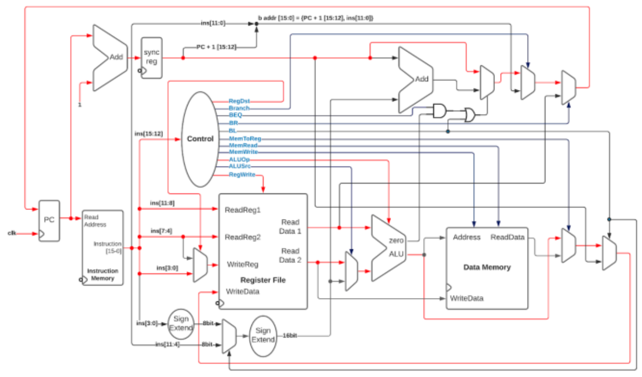


Fig. 7. R-type instruction datapath with relevant pathways outlined in red.

The R-type instructions send the opcode to the control module which determines what each control signal should be set to. It sets the RegDst and RegWrite to a logic high and determines the appropriate ALU opcode to be sent to the ALU. The RegDst signal selects which 4 bits in the instruction to pass into the destination register input based on what instruction type it is. The RegWrite signal tells the register file to write the input WriteData to the destination register when the signal is a logic high. The register file reads the source, transition, and destination registers from the instruction which then outputs the source and transition registers' data to the ALU. The ALU decodes the ALU opcode to determine which ALU operation should be performed on the inputs. The ALU output is then written back to the register file. This happens while the program counter is incremented by 1.

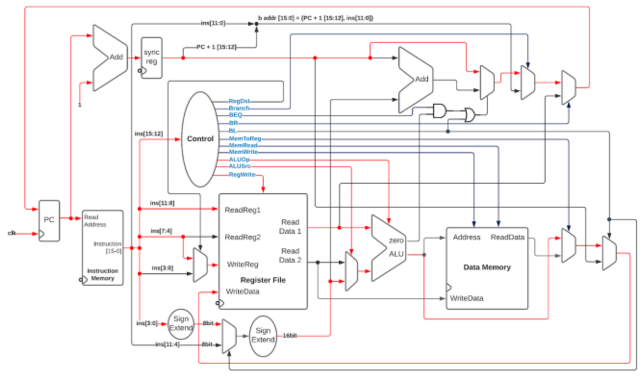


Fig. 8. I-type instruction datapath with relevant pathways outlined in red.

The I-type instruction is the same except it replaces the transition register with the immediate field. The control generates a logic high for the ALUSrc and RegWrite signals. The immediate field is sign extended to a 16-bit representation which is then passed into the ALU. A mux is used to select whether to read the second register's data or the sign-extended immediate field to pass into the ALU. It selects the immediate field when the ALUSrc control signal is logic high.

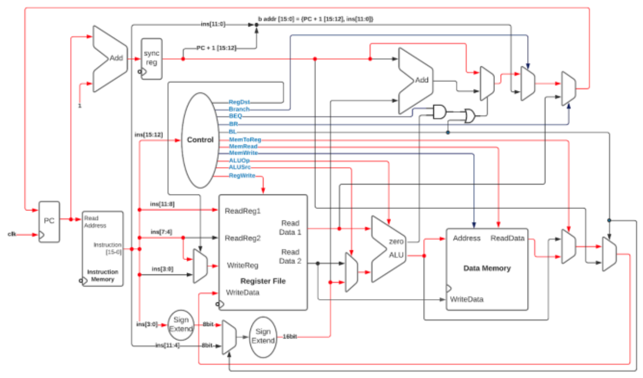


Fig. 9. Load instruction datapath with relevant pathways outlined in red.

The load instruction uses the same branching flow as the instructions above. However, it now uses the data memory. The control sets a logic high for the MemToReg, MemRead, ALUSrc, and RegWrite control signals. The immediate field is added to the source register to determine which data memory address should be read. The output of the data memory is then written back to the register file with a mux to select the data memory read output over the ALU output. It selects the data memory output when the MemToReg signal is logic high. The MemRead tells the data memory to read data at the input address when it is a logic high.

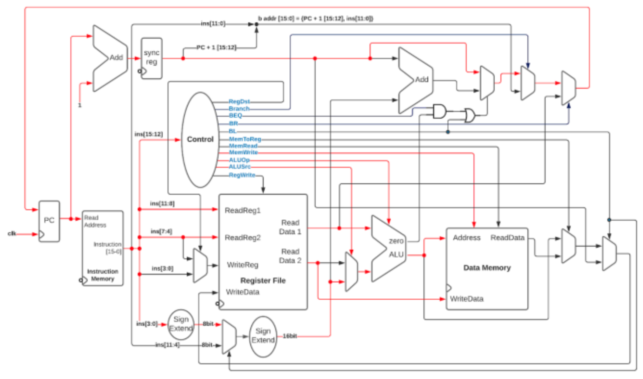


Fig. 10. Store instruction datapath with relevant pathways outlined in red.

The store instruction is similar to the load instruction, except it writes to the data memory instead of reading from it. The control sets a logic high for the MemWrite which tells the data memory to write the input data to the data memory at the given address when it is a logic high.

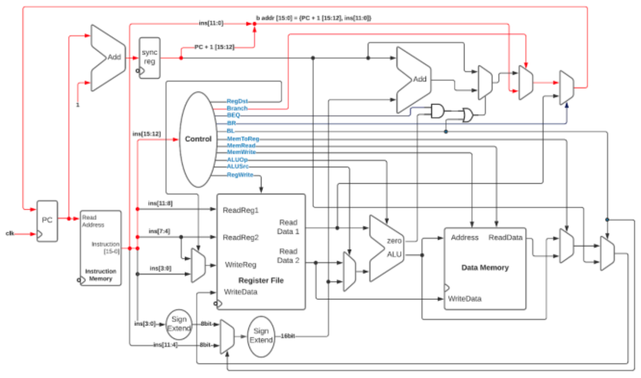


Fig. 11. Branch instruction datapath with relevant pathways outlined in red.

The branch instruction was modified so the lower 12 bits of the instruction are concatenated with the upper 4 bits of the next program counter plus 1.

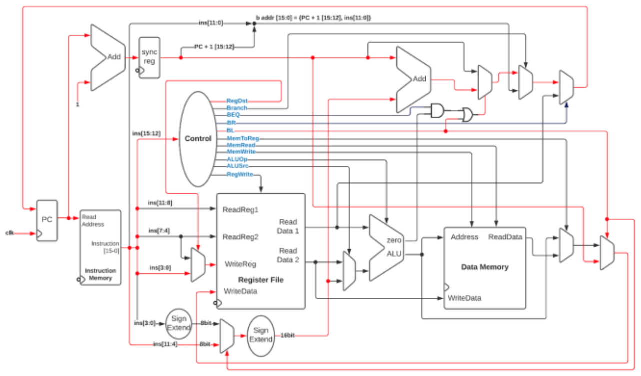


Fig. 12. Branch Link instruction datapath with relevant pathways outlined in red.

We made many modifications to incorporate the branch link and branch return instructions. For the bl instruction, the program counter plus 1 was not selected in the branching muxes to be passed back to the program counter, but was instead passed to a mux that selects what data should be written to a register. This handles the linking aspect of the bl instruction. The branching is done by passing an 8-bit address offset which gets added to the program counter plus 1. This shares the same adder hardware with the beq instruction. The branching mux that determines if that adder's output should be sent forward to the program counter was modified so that it will select its output iff the BEQ control signal and the zero flag from the ALU are both logic high or the newly added BL control signal is logic high. This handles the branching aspect of the bl instruction.

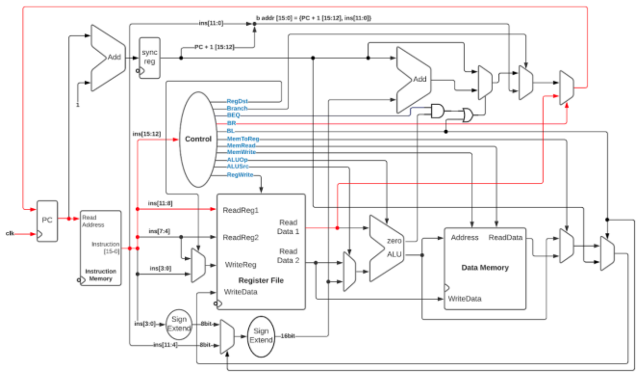


Fig. 13. Branch Return instruction datapath with relevant pathways outlined in red.

For the br instruction, a new branching mux was made which is inserted between the other branching muxes and the program counter. This uses a new BR control signal which selects to pass the data from the source register to the program counter. This branches directly to the instruction address stored in the source register. Typically, this instruction is only used with the link register to combine with the bl instruction.

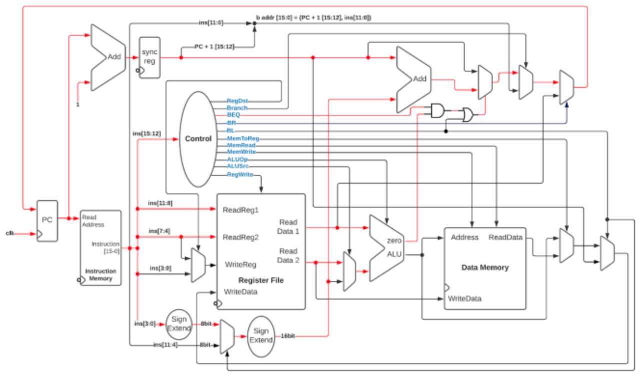


Fig. 14. Branch Equal instruction datapath with relevant pathways outlined in red.

The branch equal instruction uses the ALU to subtract the transition register from the source register. The ALU sets the zero output flag to a logic high if the subtraction result is equal to zero. The immediate field is sign extended to 16-bits and is passed to a branching offset adder to get summed with the program counter plus 1. If the control signal BEQ and the zero flag are both logic high, then a branching mux selects the branching offset adder output to be passed to the program counter.

The next modification was to have the program counter's next address calculated in an adder that was optimized to add 1. This is due to the program counter being halfword addressable, so each 16-bit instruction is stored at an incremented address of 1. The output of that adder was registered to update at the negative edge of the clock. This was due to the FPGA synthesis of the design needing the other memory blocks to be read on a clock edge. This registered adder output allowed it to synchronize with the components that were forced to be clocked. The instruction memory and write to register file were modified to update on the negative edge of the clock, while reading and writing to the data memory were set to update on the positive edge of the clock. These had to be clocked to allow the design to be synthesized. Also, the decision to choose some of the components to update on different clock edges was to allow for it to operate without any hazards. This produced a pipeline in the design without the need for pipeline stage registers.

F. *Assembling the Instructions*

Modern software developers write programs in high level languages like C# or Java, which gets optimized and translated into machine code by the compiler whereas assembly gets assembled into machine code by an assembler. While assembly languages give the programmer extensive control, it severely limits productivity. Thus, inline assembly is only used for small parts of a program that requires critical execution requirements. For this project, a custom C++ assembler was devised to convert the assembly instructions from a .txt file into another with its corresponding binary halfwords. This output .txt file is then directly initialized to the instruction memory without tedious copy pasting, allowing for swapping between different programs on the CPU quickly. It utilizes string parsing algorithms to translate the input text into machine code, which must follow very strict formatting. Because software was not the primary focus of the project, the compiler is limited in its functionality. It does not support comments and all literal inputs must be in base 10. Furthermore, since the beq instruction only supports 4-bit signed branching, it is up to the programmer to implement auxiliary labels for branches larger than -8 or +7. Similarly, the bl instruction and b support labels sizes of 8 and 12 bits respectively.

IV. SIMULATION AND SYNTHESIS

A. *Example Assembly Programs*

The project features some simple assembly programs to test the functionality of the CPU.

```
start:    add   r3, r_zero, r12;
          beq   r3, r_zero, end;
          addi  r1, r_zero, 1;
          addi  r0, r_zero, 0;
          addi  r4, r_zero, 0;
loop:     add   r2, r1, r0;
          add   r0, r_zero, r1;
          add   r1, r_zero, r2;
          addi  r4, r4, 1;
          beq   r4, r3, end;
          b     loop;
end:      add   r0, r_zero, r2;
          b     end;
```

Fig. 15. Assembly code for finding the nth term of the fibonacci sequence with for loops. The first term is defined to start at 1.

```
start:    addi  r4, r_zero, -1;
          lsr   r4, r4, 7;
          lsr   r4, r4, 4;
          and   r0, r4, r12;
          lsr   r12, r12, 5;
          and   r1, r4, r12;
          bl    multiply;
end:      b     end;

multiply: str   lr, sp, 0;
          addi  sp, sp, 1;
```

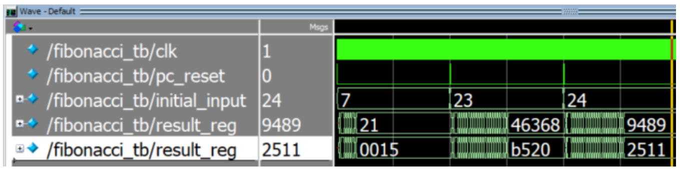
Fig. 16. Assembly code for multiplying the 5-bit input[9:5] * input[4:0]. The built-in sign extension for the top 6 bits were not used for this program.

```
start:    add   r1, r_zero, r12;
          addi  r4, r_zero, 1;
          addi  r0, r_zero, 0;
          bl    func;
          b     end;
func:     str   lr, sp, 0;
          str   r1, sp, 1;
          addi  sp, sp, 2;
          slt   r2, r4, r1;
          beq   r2, r_zero, rec_end;
          addi  r1, r1, -1;
          bl    func;
          add   r0, r1, r0;
          addi  r0, r0, 1;
          b     end_func;
rec_end:  addi  r0, r_zero, 1;
end_func: addi  sp, sp, -2;
          ldr   r1, sp, 1;
          ldr   lr, sp, 0;
          br    lr;
end:      b     end;
```

Fig. 17. Assembly code for finding the arithmetic sequence (1+2+3+...+n) with recursion. The derived general solution is n*(n+1)/2.

B. *CPU Simulation and FPGA Operation*

The following figures show the function of the assembly programs listed above. For the fibonacci, an input of 24 yields the max output before overflow. For the following FPGA figures the leftmost digit display is there only for debugging the pushbuttons. The 16-bit output is presented as 4 hex digits.



	Msgs			
/fibonacci_tb/clk	1			
/fibonacci_tb/pc_reset	0			
/fibonacci_tb/initial_input	24	7	23	24
/fibonacci_tb/result_reg	9489	21	46368	9489
/fibonacci_tb/result_reg	2511	0015	b520	2511

Fig. 18. Waveform simulation of the CPU running the fibonacci sequence test assembly program.

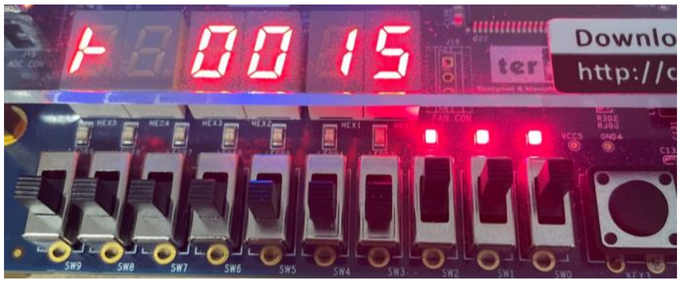


Fig. 19. FPGA operation of the 7th fibonacci term.

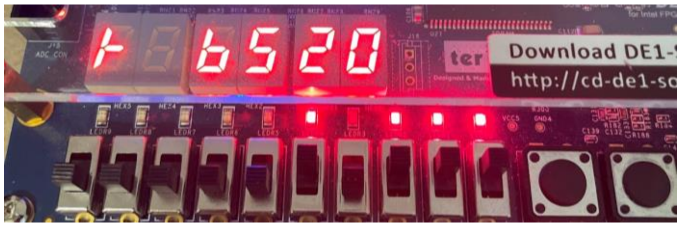


Fig. 20. FPGA operation of the 23rd fibonacci term, which is the max value before overflow.

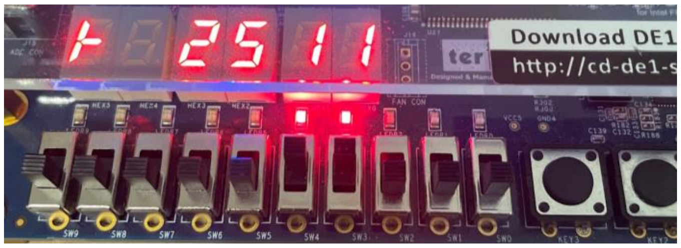
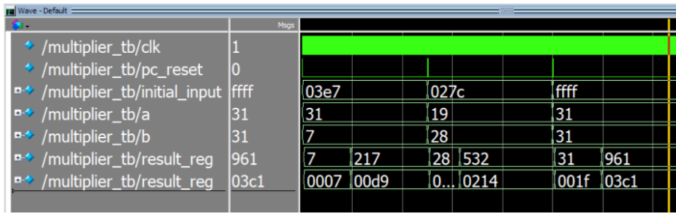


Fig. 21. FPGA operation of the 24th fibonacci term, which overflowed since it requires 5 hex digits to represent.



```
/multiplier_tb/clk            1
/multiplier_tb/pc_reset       0
/multiplier_tb/initial_input  ffff   03e7        027c        ffff
/multiplier_tb/a              31     31          19          31
/multiplier_tb/b              31     7           28          31
/multiplier_tb/result_reg     961    7    217    28   532    31    961
/multiplier_tb/result_reg     03c1   0007 00d9   0... 0214   001f  03c1
```

Fig. 22. Waveform simulation of the CPU running the unsigned multiplier test assembly program.

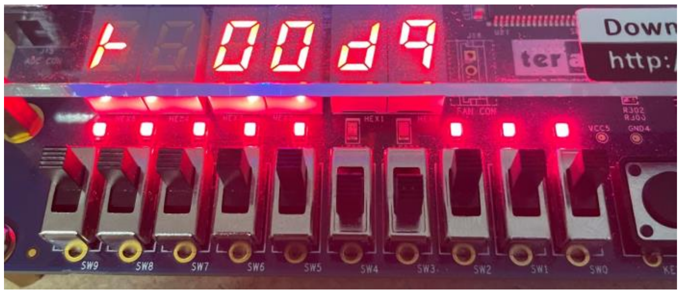


Fig. 23. FPGA operation of 7x7.

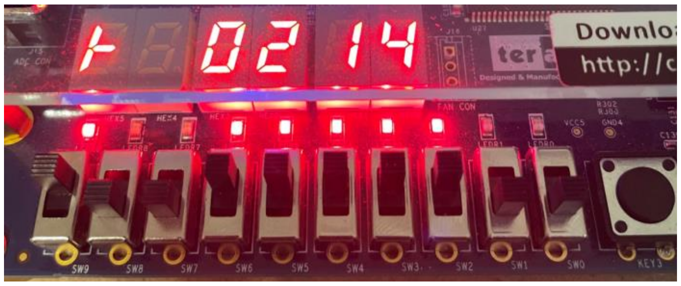


Fig. 24. FPGA operation of 19x28.

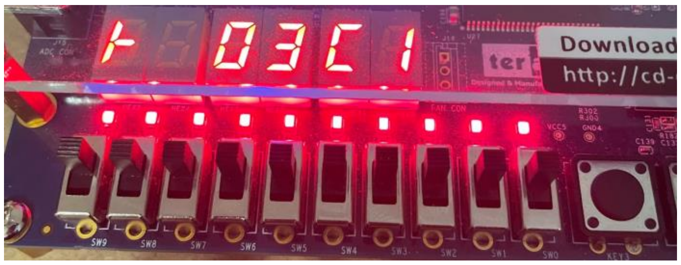
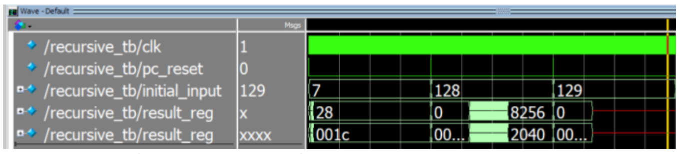


Fig. 25. FPGA operation of 31x31.



```
/recursive_tb/clk            1
/recursive_tb/pc_reset       0
/recursive_tb/initial_input  129    7       128           129
/recursive_tb/result_reg     x      28      0      8256   0
/recursive_tb/result_reg     xxxx   001c    00...  2040   00...
```

Fig. 26. Waveform simulation of the CPU running the recursive sum test assembly program.

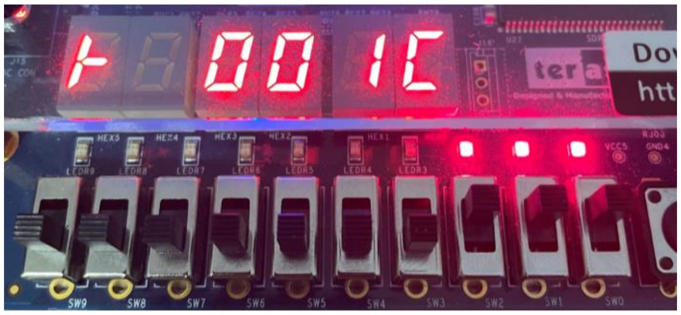


Fig. 27. FPGA operation of recursive sum for n = 7.

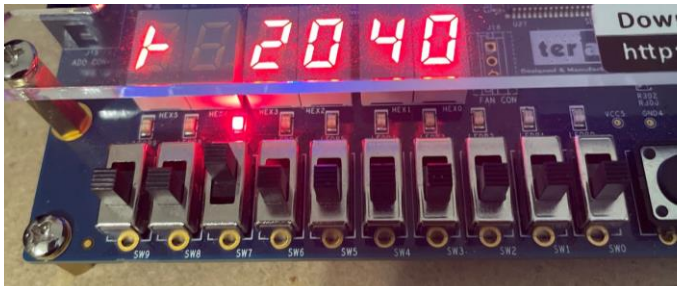


Fig. 28. FPGA operation of recursive sum for n = 128.

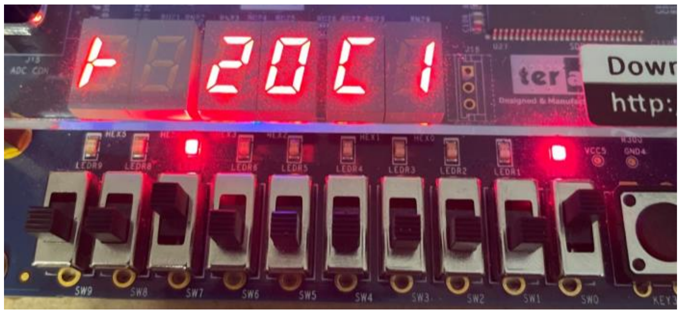


Fig. 29. FPGA operation of recursive sum for n = 129.

The results of each test case for each test assembly program matched and yielded correct values except for the third test input for the recursive sum test program. This exception was an interesting observation. The third test case provided don't-care values in the ModelSim simulation waveform. This is as expected due to the stack overflowing at this value. However, when synthesized on the DE1-SoC FPGA, the stack did not overflow and instead computed the correct result. We speculate that even though our program is only utilizing 1024 halfwords of data memory, the FPGA allocated more space on its block RAM for the data memory module than anticipated.

C. Synthesis Report

Timing:

- Clock Speed: 66.67 MHz (15 ns)
- Setup Time Slack: 0.274 ns
- Hold Time Slack: 0.223 ns

Area:

- Logic utilization: 484 / 32,070 (2 % ALMs)
- Total Registers: 279
- Total Block Memory: 32,800 / 4,065,280 (< 1 % bits)

Power:

- Total Power Dissipation: 432.32 mW
- Dynamic Power Dissipation: 5.67 mW
- Static Power Dissipation: 411.38 mW
- I/O Power Dissipation: 15.27 mW

Fig. 30. FPGA DE1-SoC synthesis for fast 1100mW 85℃ operation.

The synthesis was run on Intel Quartus Prime with aggressive speed optimizations at the cost of area. Due to the pipelined (albeit unorthodox) stages being on different clock edges, we were able to reduce the clock period to 15 ns while still maintaining positive hold time and setup time slack. Furthermore, the logic utilization was low relative to the FPGA and this CPU only had 279 registers, 256 of which were used for the 16 halfword latches in the registers file. Finally, the total block memory is consistent with the 1024-halfword sized instruction and data memory. The power report was generated with the default settings, which simulates 12.5% input toggling. This may not be accurate with the actual CPU operation, however, is adequate for a first order approximation.

V. CONCLUSION

The overall result of the 16-bit Harvard CPU architecture design was successful. Throughout the implementation phase, there were several instances where we had to redesign portions of the architecture due to incompatibilities between components and the FPGA requiring the block memory to be read synchronously. There are many ways that this project can be built upon, such as implementing full pipelining support or a floating-point coprocessor. This project provided valuable experience in computer architecture design as well as verilog and synthesis. As computer engineering majors, the experience and skills gained from this project will be very useful in our future careers, especially in the hardware industry.

REFERENCES

[1] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The hardware/software interface*. Cambridge, MA: Morgan Kaufmann Publishers, an imprint of Elsevier, 2021.