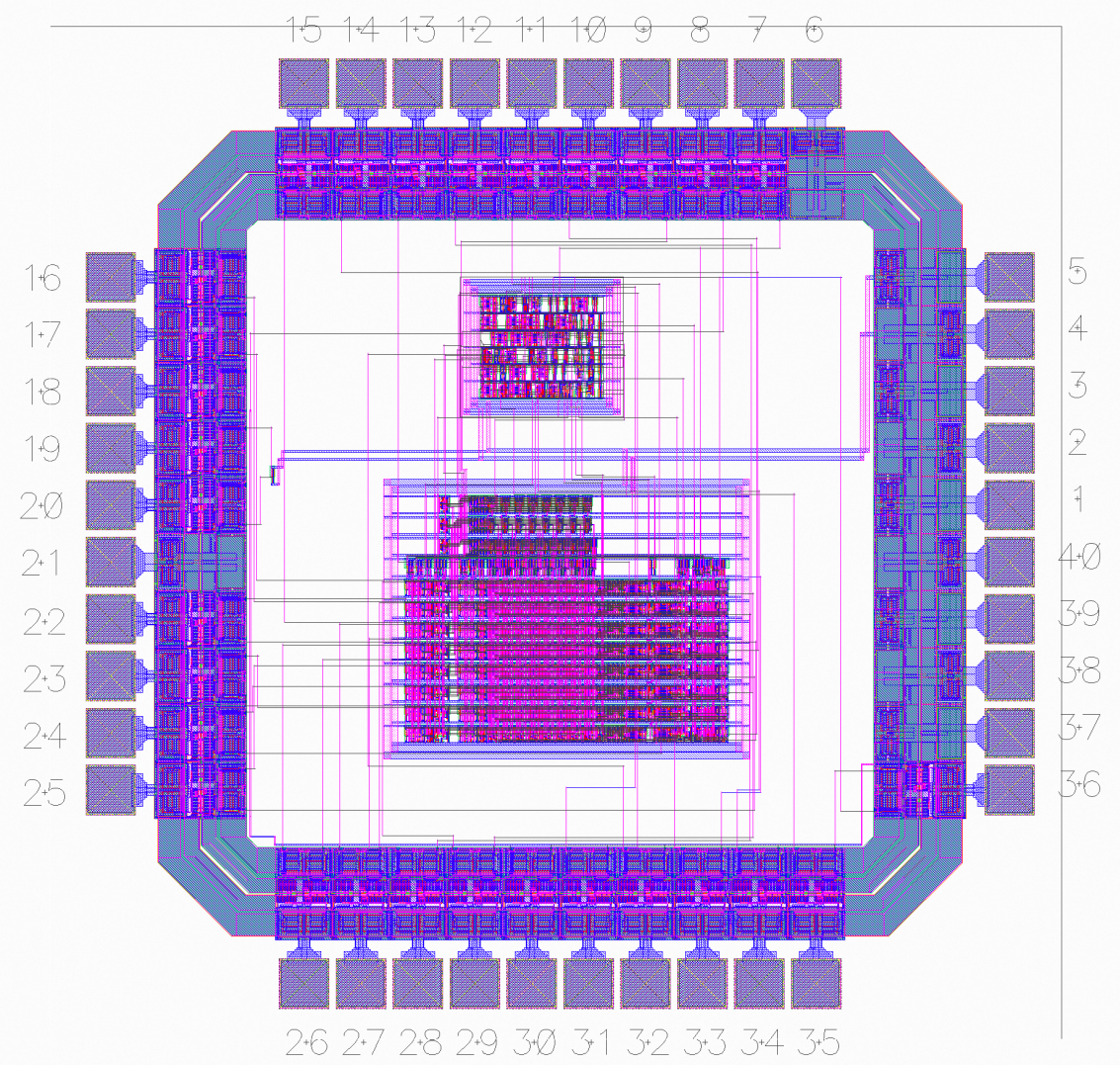

E190AT Final Project Report:
Harvey Mudd Miniature Machine Minified

Erik Meike, Caleb Norfleet, and Kaveh Pezeshki

April 2019

1 Introduction

CSCI005, Introduction to Computer Science, and CSCI042, Principles and Practice of Computer Science, teach many Harvey Mudd freshman low-level computer architecture through the Harvey Mudd Miniature Machine (HMMM) ISA ("Documentation for HMMM"). While the HMMM ISA provides an effective tool to learn assembly language, it can be difficult for students to connect the Python HMMM simulator to the physical computers they interact with daily.

This report presents a physical implementation of a subset of the HMMM ISA which we have named the Harvey Mudd Miniature Machine Minified (HMMMM), designed for the $0.6\mu m$ AMI process ($\lambda = 0.3\mu m$) on a 1.5×1.5mm die. We hope that a phyical implementation of the HMMM ISA will help connect high-level computer science concepts to real-world devices.

HMMMM implements a 15-bit-instruction dual-cycle CPU, which operates on 8-bit data words. The processor requires an external SRAM, as shared instruction and data memory, accessed over an 8-bit address bus. In the context of a larger system, users will be able to program the system through an external bank of switches.

2 HMMMM Design

2.1 Overall Architecture

HMMMM is a simple 15-bit-instruction, 8-bit-word processor, with eight general purpose registers. The supported instruction set is described in Figure 1. In order to simplify the shared instruction and data memory without a pipelined multicycled architecture, instructions take two cycles to execute (with branch instructions, which only require one cycle, as an exception), allowing for separate clock cycles for instruction execution and memory write-back.

Instruction Type	Instruction	Encoding				
		[14:11]	[10:8]	[7:5]	[4:2]	[1:0]
Data Processing	copy	0100	<write reg>	xxx	<read reg>	xx
Data Processing	add	0110	<write reg>	<read reg 1>	<read reg 2>	xx
Data Processing	sub	0111	<write reg>	<read reg 1>	<read reg 2>	xx
Data Processing	neg	0101	<write reg>	xxx	<read reg>	xx
Memory	loadr	0011	<data reg>	xxx	<addr reg>	xx
Memory	storer	0010	xxx	<data reg>	<addr reg>	xx
Immediate	setn	0001	<write reg>	<immediate[7:0]>		
Branch	jumpn	110x	xxx	<immediate[7:0]>		
Branch	jumpr	111x	<addr reg>	xxxxxxxx		
Branch	jeqzn	1000	<data reg>	<immediate[7:0]>		
Branch	jneqzn	1001	<data reg>	<immediate[7:0]>		
Branch	jgtzn	1010	<data reg>	<immediate[7:0]>		
Branch	jltzn	1011	<data reg>	<immediate[7:0]>		
NOP	nop	0000	xxx	xxxxxxxx		

Figure 1: The HMMMM Instruction Set. All register addresses are three bits wide.

The processor exposes the 8-bit address bus, 15-bit data bus, as well as a memory write control signal for the SRAM. The SRAM is assumed to be always enabled, and is assumed to be in read mode when memory write is low.

2.2 Datapath Design

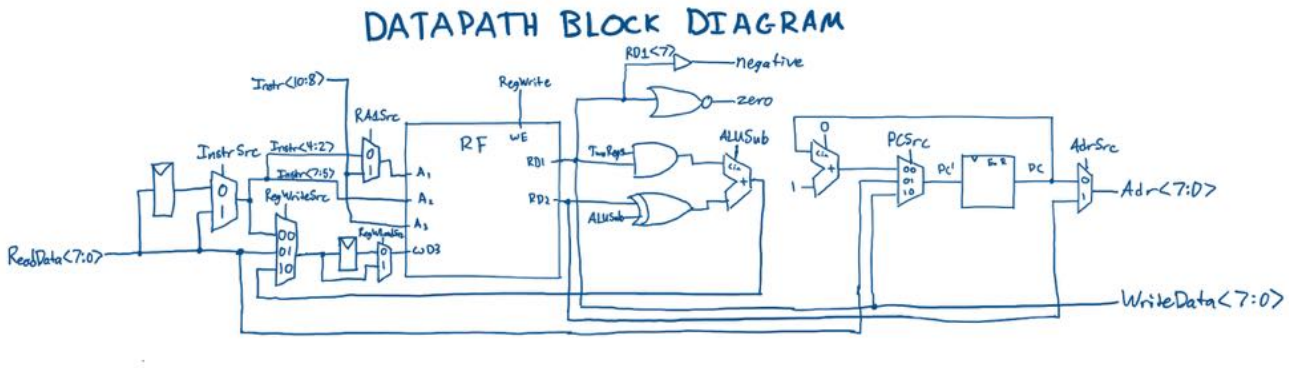

Figure 2: Datapath Block Diagram

The chip is split up into two main blocks: the datapath and the controller. The datapath, with a schematic shown in Figure 2, and HDL in Appendix A, handles the processing and routing of data, and includes the register file and the program counter. The datapath layout is a custom block, and it is split into 12 rows. The bottom 8 rows handle the 8-bit data stream, the top 3 rows contain the register file address decoder, and the row in between contains the zipper, where control signals are routed to the datapath from the controller.

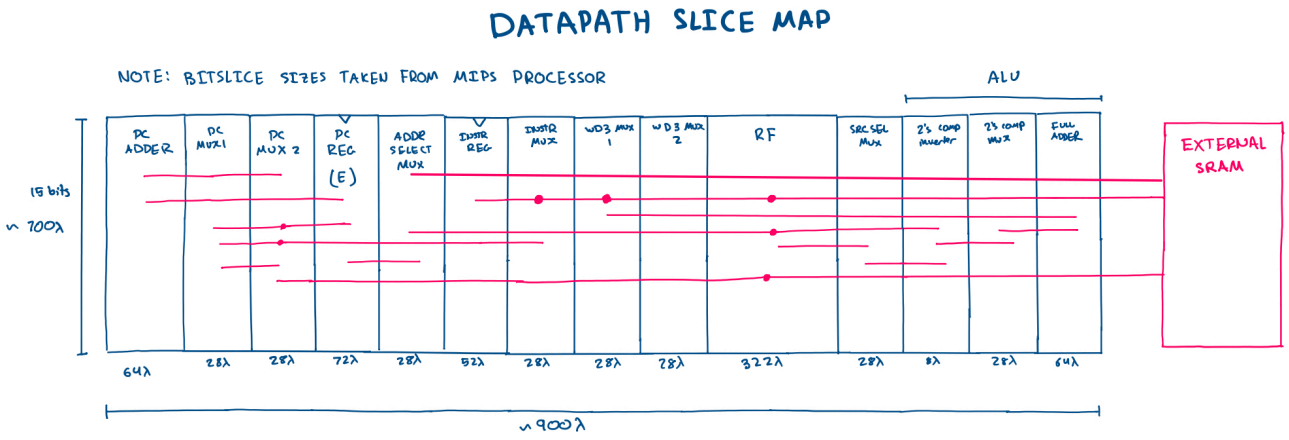

Figure 3: Proposed Datapath Slice Map

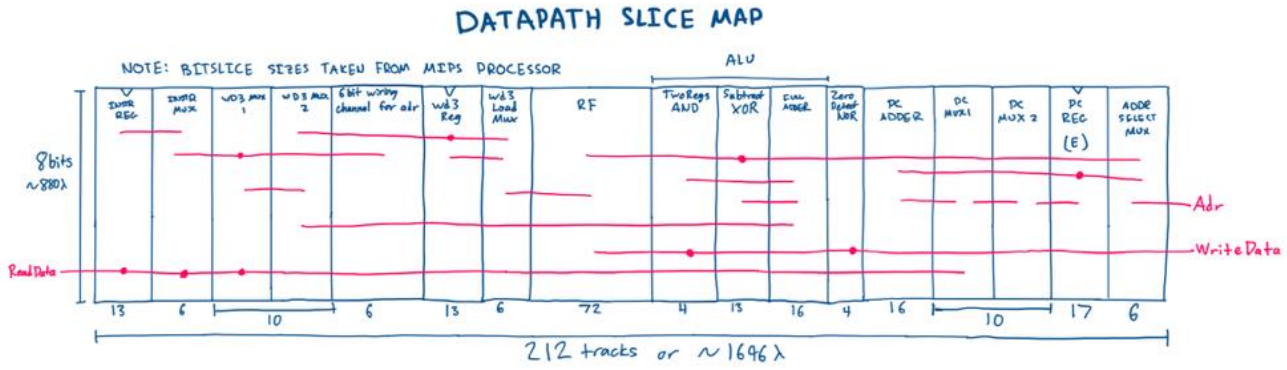

Figure 4: Final Datapath Slice Map

The slice map (Figures 3 and 4) outlines the placement of leaf cells and the routing of signals for each of the bottom 8 rows in the datapath. By summing the width of the leaf

cells one can estimate the total size of the datapath (See Figures 7 and 8 for the full chip floorplan).

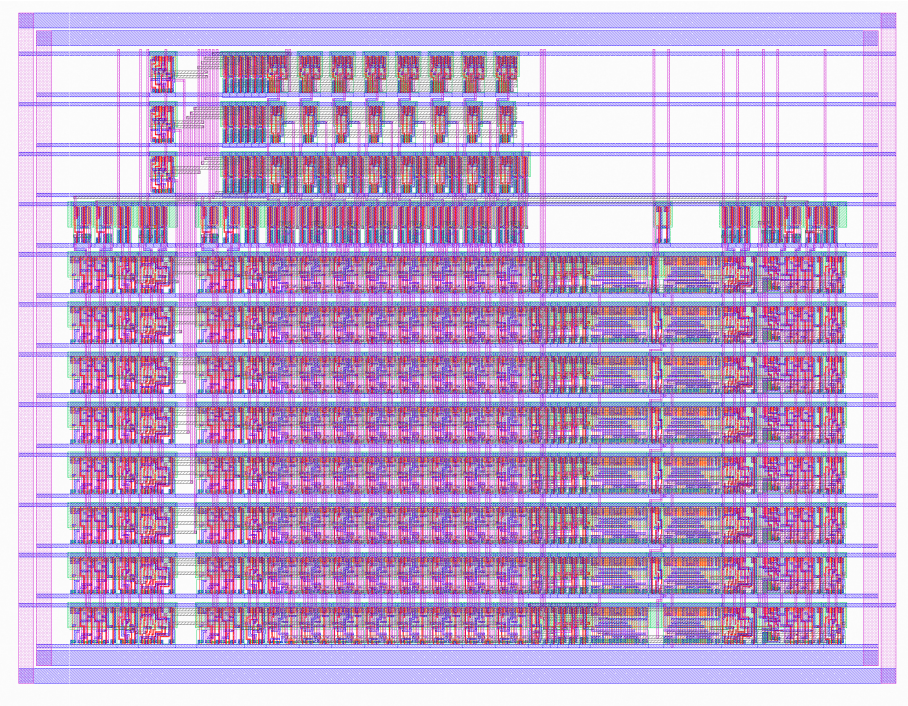

Figure 5: The datapath layout. A version with labelled cells is available in Appendix E.2.

2.3 Controller Design

The controller processes an input instruction to select multiplexer and ALU modes throughout the datapath. The HDL, available in Appendix A, is synthesized into a schematic and layout.

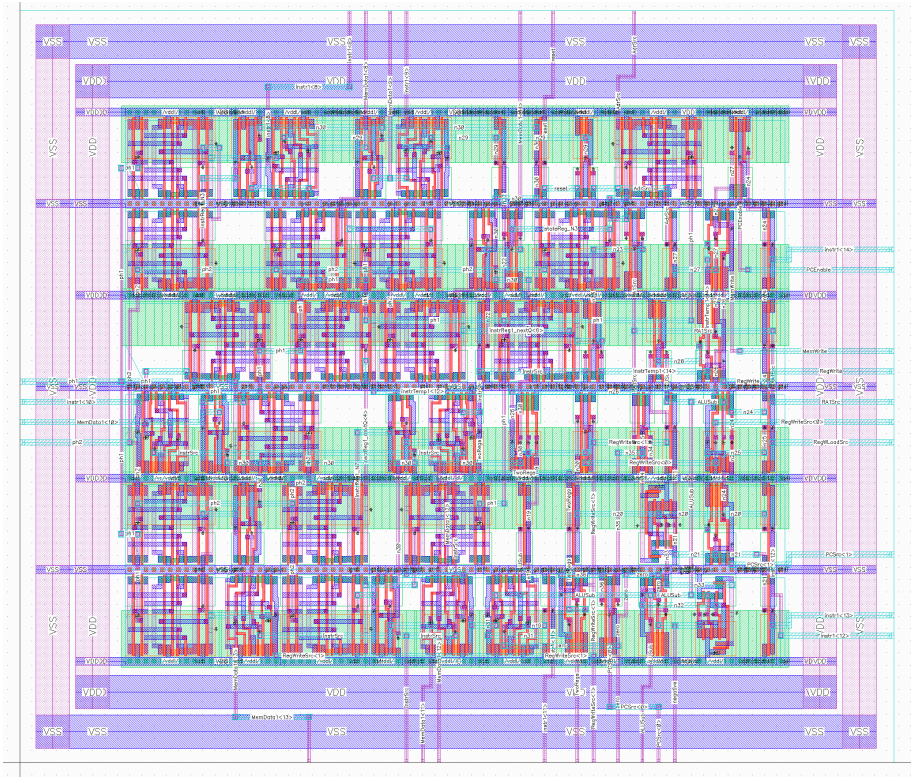

Figure 6: The controller layout. A version with labelled cells is available in Appendix E.1.

2.4 Floorplan

The proposed and completed floorplans are given in Figure 7 and Figure 8, respectively. Notable is a $\approx 2\times$ underestimate of datapath width in the proposed floorplan. This was due to a mis-calculation of standard cell width in the project proposal. No major changes to the datapath were made between the proposal and final layout.

This disparity is most clearly seen when examining the slice plans. The proposed and final datapath slice maps have an identical configuration, with the only difference the estimated and final width.

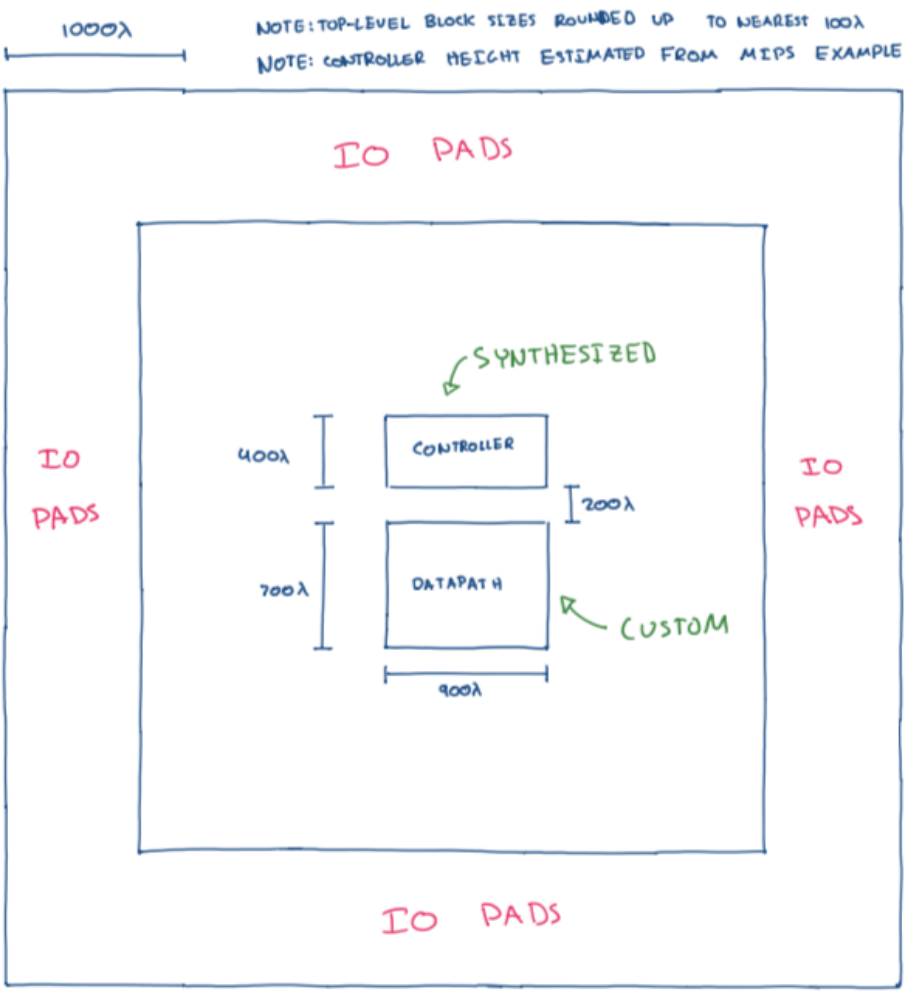

Figure 7: Proposed Floorplan

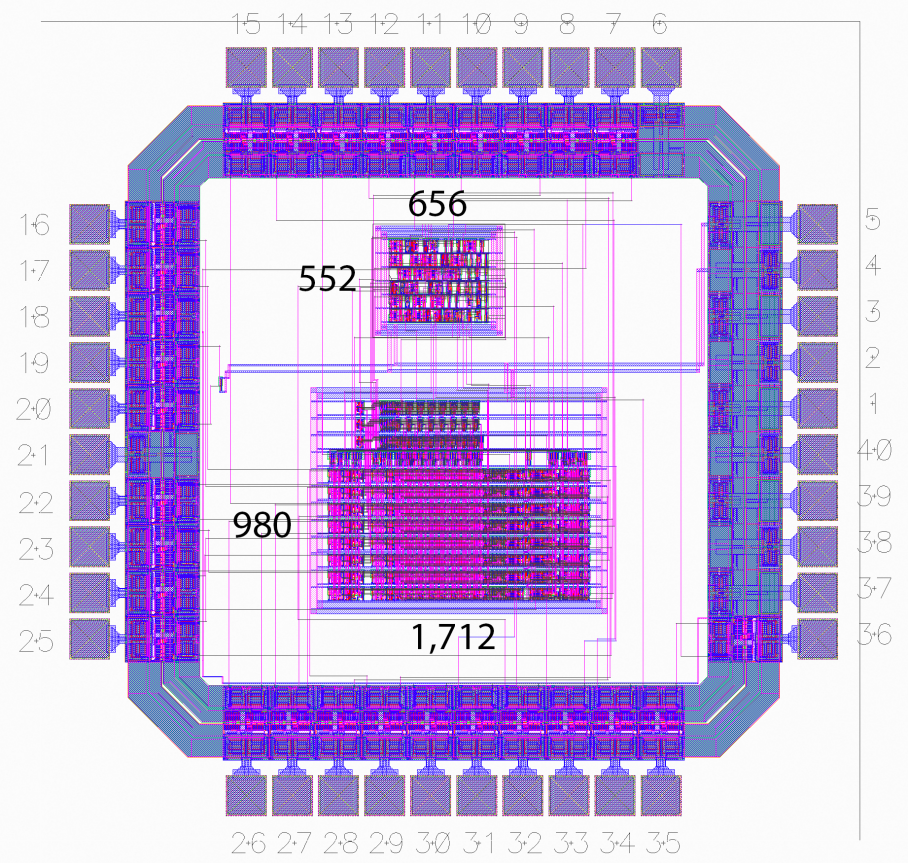

Figure 8: Final Floorplan, Units in λ

2.5 Pinout

MOSIS packages the die in a 40 pin DIP. The package pinout in tabular form is provided in Figure 9. Figure 10 provides a graphical view of the device pinout. Pins 19 and 20 are tied to an inverter as a basic manufacturing test.

Pin Number	Pin Function	Pin Direction
1	vdd	power
2	gnd	power
3	vdd	power
4	gnd	power
5	vdd	power
6	gnd	power
7	reset	input
8	clock phase 1	input
9	clock phase 2	input
10-17	address[0:7]	output
18	memwrite	output
19	test_in	input
20	test_out	output
21	gnd	power
22-36	memdata[0:14]	bidirectional
37	vdd	power
38	gnd	power
39	vdd	power
40	gnd	power

Figure 9: The HMMMM pinout. test_in and test_out are the input to and output from an inverter independent from the processor.

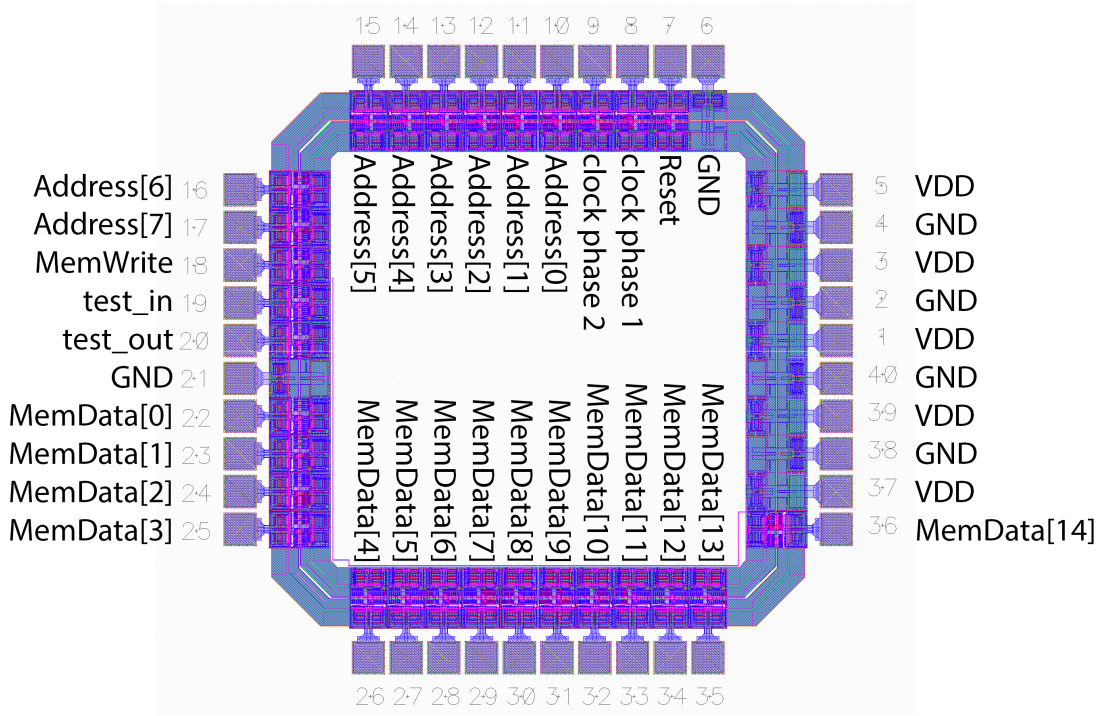

Figure 10: A graphical view of the HMMMM pinout

3 Verification

The table in Figure 11 summarizes the verification status of HMMMM.

Verification Step	Status
HDL Passing Testbench	Complete
Controller Passing DRC	Complete
Controller Passing LVS	Complete
Controller Passing Testbench	Complete
Datapath Passing DRC	Complete
Datapath Passing LVS	Complete
Datapath Passing Testbench	Complete
Chip Passing DRC	Complete
Chip Passing LVS	Complete
Chip Passing Testbench	Complete
GDS Re-import	Complete
GDS Verification	Complete

Figure 11: HMMMM Verification Status.

3.1 HDL and Schematic Verification

We created a self-checking SystemVerilog testbench and SRAM for controller and datapath verification[1]. On simulation initialization, the SRAM loads a 32-instruction program[2], and initializes address `0x40` to `0x05`. This program tests every HMMMM instruction, including less than zero, equal to zero, and greater than zero for all conditional branches.

Successful program execution will lead to the value `0x2D` written in address `0x29`. If this is not observed (i.e. if the value `0x00` appears in address `0x29`), then then the processor has malfunctioned. We substituted exported Verilog in for the controller and datapath when designing the controller and datapath schematic, and have ensured that the final design passes the testbench.

3.2 Layout Verification

After finishing each subsection of the chip, we performed LVS (Layout vs Schematic) by extracting the schematic from our layout automatically and then comparing this to our original schematic. This allowed us to test each cell's final functionality and isolate issues before they became harder to debug later in the design process. We performed this on each schematic level on the chip from leaf cell to the full chip. We finally exported GDS of our

[1]The SystemVerilog source is available in Appendix A.1 and A.2
[2]The testvectors are available in Appendix A.3

chip and ran LVS after re-importing the final fabrication files to ensure that no parts of the chip were broken during the export process.

3.3 Post-Fabrication Test Plan

We aim to produce not just a functional chip, but a full demonstration system implementing a functional chip. This demonstration and testing system will provide:

- Software-defined SRAM and processor clocks through a Lattice FPGA with onboard non-volatile configuration memory. This allows for non-volatile program storage.
- All HMMMM I/O accessible from 0.1" headers and tri-state buffers
- A digitally-controlled variable-voltage processor power supply, implemented through an LM317 and digital potentiometer
- A shunt resistor to measure HMMMM current requirements, and an associated ADC
- Low-voltage level shifters for all processor interfaces
- LED arrays for address and data bus visualization
- A switch bank for hand-programming the SRAM

Initial testing will involve loading the HMMMM program described in Section 3.1 onto the FPGA SRAM, and testing whether the physical chip correctly executes the program. If this test is successful, we will investigate the IV characteristics of the processor, including transistor response to low voltages. If this test fails, we will follow the following debugging process:

1. Check resistance between vdd and gnd. A short here indicates a completely nonfunctional chip. If this occurs, we will try re-testing with another chip
2. Check inverter behavior across test pins 19 and 20. This is a simple test of the fabrication process, and a failure here indicates a larger problem with our layout. If this occurs, we will try re-testing with another chip
3. Debugging steps past the above points are contingent on the observed symptoms

Finally, we will implement simple assembly programs seen in CSCI005 and CSCI042 to use the device as a demonstration tool.

4 Logistics

4.1 Design Time

The design time breakdown is illustrated in Figure 12.

Design Component	Design Time (Hours)	Verification Time (Hours)
Testbench HDL	2	1
SRAM HDL	1.5	0.5
Datapath HDL	10	2
Datapath Schematic	6	1.5
Datapath Layout	9	4
Controller HDL	4	3
Controller Schematic	7	2
Controller Layout	2	1
Chip Schematic	3	4
Chip Layout	2	1
Chip Export	1	0.5
Leaf Cell	1	0.5

Figure 12: Design time breakdown for the HMMMM processor

4.2 File Locations

All human-readable source files are included in the appendix to this report. Design and testbench HDL, testvectors, GDS[3], and PDF[4]. of this report are available on the project GitHub[5].

Cadence design files are accessible in the repository under directory `cadence_design_files`. Design components are available under the following libraries and cells described in Figure 13.

[3]https://github.com/VLSIProject2019/vlsi-2019/exported_chip.gds
[4]https://github.com/VLSIProject2019/vlsi-2019/final_report.pdf
[5]https://github.com/VLSIProject2019/vlsi-2019

Design Component	Library	Cell
Full Chip	HMMM	chip
Chip (no padframe)	HMMM	top
Padframe	HMMM	padframe
Datapath	HMMM	datapath
Controller	HMMM_Synth	controller

Figure 13: Cadence design file locations

5 References

"Documentation for HMMM (Harvey Mudd Miniature Machine)" *Harvey Mudd College*, Fall, 2018. Web.

Appendix

A HDL Source Code

```
// Microprocessor for E190AT: VLSI Design Project
// Authors: Erik Meike, Caleb Norfleet, & Kaveh Pezeshki
// Created Spring 2019

module top (input  logic       ph1, ph2, reset,
                               output logic       MemWrite,
                               output logic [7:0]  Adr,
                               input  logic [14:8] MemData1,
                               inout  logic [7:0]  MemData2);

        logic PCEnable, AdrSrc, RA1Src, InstrSrc, RegWrite;
        logic TwoRegs, ALUSub, negative, zero, RegWLoadSrc;
        logic [1:0]  PCSrc, RegWriteSrc;
        logic [7:0]  WriteData;
        logic [14:8] instr1;

        // tristate for handling write data
        assign MemData2[7:0]  = (MemWrite ? WriteData : 8'bz);

        controller c(ph1, ph2, reset, negative, zero, RegWLoadSrc,
                                      RA1Src, PCEnable, AdrSrc, InstrSrc, RegWrite,
                                      TwoRegs, ALUSub, PCSrc, RegWriteSrc, MemWrite,
                                      MemData1, instr1);

        //datapath dp(Adr, negative, zero, MemData2[7:0], ALUSub,
        //            AdrSrc, instr1[10:8], InstrSrc, MemWrite,
        //            PCEnable, PCSrc, RA1Src, RegWLoadSrc, RegWrite,
        //            RegWriteSrc, TwoRegs, ph1, ph2, reset );

        datapath dp(ph1, ph2, reset, PCEnable, AdrSrc, InstrSrc,
                                     RA1Src, RegWrite, MemWrite, TwoRegs, ALUSub,
RegWLoadSrc,
                                     PCSrc, RegWriteSrc, instr1[10:8], MemData2, WriteData,
                                     Adr, negative, zero);
endmodule

module datapath (input  logic      ph1, ph2, reset,
                                   input  logic      PCEnable, AdrSrc, InstrSrc, RA1Src,
RegWrite,
                                   input  logic      MemWrite, TwoRegs, ALUSub,
RegWLoadSrc,
                                   input  logic [1:0]  PCSrc, RegWriteSrc,
                                   input  logic [10:8] instr1,
```

```
                                input  logic [7:0]  MemData2,
                                output logic [7:0]  WriteData,
                                output logic [7:0]  Adr,
                                output logic        negative, zero);

logic[7:0]  PC, PCNext, PCPlus1;
logic[7:0]  Result, SrcA, SrcB, Imm, WD3;
logic[7:0]  WD3Temp, WD3Temp2, RD1, RD2;
logic[2:0]  RA1, RA2, WA3;
logic[7:0]  instrTemp2, instr2;

// next PC logic
adder   #(8) pcAdd(PC, 8'b1, 1'b0, PCPlus1);
flopenr #(8) pcReg(ph1, ph2, reset, PCEnable, PCNext, PC);
mux3    #(8) pcMux(PCPlus1, Imm, RD1, PCSrc, PCNext);

// data memory
mux2 #(8) adrMux(PC, RD2, AdrSrc, Adr);
assign WriteData = RD1;

// instruction handling
flopr #(8) instrReg2(ph1, ph2, reset, MemData2, instrTemp2);
mux2  #(8) instrMux2(instrTemp2, MemData2, InstrSrc, instr2);

// register read/write logic
mux3 #(8) wd3Mux(Imm, MemData2[7:0], Result, RegWriteSrc, WD3Temp);
flop #(8) wd3Reg(ph1, ph2, reset, WD3Temp, WD3Temp2);
mux2 #(8) loadMux(WD3Temp2, WD3Temp, RegWLoadSrc, WD3);
regfile   rf(ph1, ph2, reset, RegWrite, RA1, RA2, WA3, WD3, RD1, RD2);
mux2 #(3) ra1Mux(instr2[7:5], instr1[10:8], RA1Src, RA1);
assign RA2 = instr2[4:2];
assign WA3 = instr1[10:8];
assign Imm = instr2[7:0];

// zero and negative for conditional branching
assign negative = RD1[7];
assign zero = ~(|RD1);

// ALU logic
assign SrcB = {8{ALUSub}} ^ RD2;
// ^same as: mux2  #(8) srcBMux(RD2, notRD2, ALUSub, SrcB);
assign SrcA = {8{TwoRegs}} & RD1;
// ^same as: mux2  #(8) srcAMux(8'b0, RD1, TwoRegs, SrcA);
```

```
        adder #(8) alu(SrcA, SrcB, ALUSub, Result);
endmodule//*/

module controller (input  logic      ph1, ph2, reset,
                                        input  logic      negative, zero,
                                        output logic      RegWLoadSrc, RA1Src,
PCEnable, AdrSrc,
                                    output logic      InstrSrc, RegWrite, TwoRegs,
ALUSub,
                                    output logic[1:0]  PCSrc, RegWriteSrc,
                                        output logic      MemWrite,
                                        input  logic[14:8] MemData1,
                                        output logic[14:8] instr1);
        logic state, stateBar, condBranch;
        logic branch, unconditional, regJumpLoc;
        logic[3:0] funct;
        logic[14:8] instrTemp1;

        // instruction handling
        flopr #(7) instrReg1(ph1, ph2, reset, MemData1, instrTemp1);
        mux2  #(7) instrMux1(instrTemp1, MemData1, InstrSrc, instr1);
        assign funct = instr1[14:11];

        // cycle clock "FSM" (0=instr read, 1=load/write back)
        // note: branch instructions only require one cycle
        flopr #(1) stateReg(ph1, ph2, reset, stateBar & ~branch, state);
        assign stateBar = ~state;

        assign PCEnable = state | branch;
        assign AdrSrc   = state;
        assign InstrSrc = ~reset & stateBar;

        // branch
        assign branch       = funct[3];
        assign unconditional = funct[2];
        assign regJumpLoc    = funct[1];
        condcheck cc(funct[1:0], negative, zero, condBranch);
        assign PCSrc = (branch & (unconditional | condBranch)) ?
                                            ((unconditional & regJumpLoc) ?
                                            2'b10 : 2'b01) : 2'b00;
        assign RA1Src = branch;

        // data processing
```

```
        assign TwoRegs = funct[1];
        assign ALUSub  = funct[0];

        // writeback
        assign MemWrite = (state & (funct == 4'b0010)) & ~reset;
        // ^Note: added &~reset so that first instruction can be loaded at init
        assign RegWrite = state & ~branch & (funct[2] | funct[0]);
        always_comb
                if(funct[2])
                        RegWriteSrc = 2'b10; // Result
                else if(funct[1])
                        RegWriteSrc = 2'b01; // ReadData
                else
                        RegWriteSrc = 2'b00; // Imm
        assign RegWLoadSrc = (funct == 4'b0011);
endmodule

module condcheck (input  logic[1:0] branchType,
                                        input  logic     negative, zero,
                                        output logic     condBranch);
        always_comb
                case(branchType) // branchType = {isZero, greaterThan}
                        2'b00: // jeqzn
                                condBranch = zero;
                        2'b01: // jneqzn
                                condBranch = ~zero;
                        2'b10: // jgtzn
                                condBranch = ~negative & ~zero;
                        2'b11: // jltzn
                                condBranch = negative; // if negative then also nonzero
                        default:
                                condBranch = 1'bx;
                endcase
endmodule

module adder #(parameter WIDTH=8)
                                        (input  logic [WIDTH-1:0] a, b,
                                                input  logic cin,
                                                output logic [WIDTH-1:0] y);
        assign y = a + b + cin;
endmodule

// note: regfile copied from MIPS8 design
```

```
module regfile #(parameter WIDTH = 8)
                (input  logic     ph1, ph2, reset,
                 input  logic     we3,
                 input  logic [2:0] ra1, ra2, wa3,
                 input  logic [WIDTH-1:0] wd3,
                 output logic [WIDTH-1:0] rd1, rd2);

          // note: can't read PC in HMMM
    // three ported register file
    // read two ports combinationally
    // write third port during phase2 (second half-cycle)

    logic [WIDTH-1:0] RAM [7:0];

    always_latch
                    if (ph2 & we3) RAM[wa3] <= wd3;

          assign rd1 = RAM[ra1];
          assign rd2 = RAM[ra2];
endmodule//*/

module mux2 #(parameter WIDTH = 8)
                                    (input  logic [WIDTH-1:0] d0, d1,
                                     input  logic           s,
                                     output logic [WIDTH-1:0] y);
          assign y = s ? d1 : d0;
endmodule

module mux3 #(parameter WIDTH = 8)
                                    (input  logic [WIDTH-1:0] d0, d1, d2,
             input  logic [1:0]      s,
             output logic [WIDTH-1:0] y);
          // note: 11 is treated as the same as 10
          assign y = s[1] ? (d2) : (s[0] ? d1 : d0);
          // alternative:
          // assign y = s[1] ? (s[0] ? 8'bx : d2) : (s[0] ? d1 : d0);
endmodule

module flop #(parameter WIDTH = 8)
                                    (input  logic ph1, ph2, reset,
             input  logic [WIDTH-1:0] d,
             output logic [WIDTH-1:0] q);
          logic[WIDTH-1:0] nextQ;
```

```
        always_latch
                if (ph1) q <= nextQ;
        always_latch
                if (ph2) nextQ <= d;
endmodule

// taken from MIPS8
module flopr #(parameter WIDTH = 8)
                                (input  logic ph1, ph2, reset,
          input  logic [WIDTH-1:0] d,
          output logic [WIDTH-1:0] q);
        logic[WIDTH-1:0] nextQ;
        always_latch begin
                if (ph1)
                        q <= nextQ;
  end
        always_latch begin
                if (ph2)
                        if (reset) nextQ <= 0;
                        else       nextQ <= d;
        end
endmodule

// taken from MIPS8
module flopenr #(parameter WIDTH = 8)
            (input  logic ph1, ph2, reset, en,
             input  logic [WIDTH-1:0] d,
             output logic [WIDTH-1:0] q);
        logic [WIDTH-1:0] d2, resetval;
        assign resetval = 0;

        mux3 #(WIDTH) enrmux(q, d, resetval, {reset, en}, d2);
        flop #(WIDTH) f(ph1, ph2, reset, d2, q);
endmodule
```

B HDL Testbench Code

B.1 Testbench

```
// Microprocessor Testbench for E190AT: VLSI Design Project
// Authors: Erik Meike, Dominique Mena, Caleb Norfleet, & Kaveh Pezeshki
// Created Spring 2019


// Set delay unit to 1 ns and simulation precision to 0.1 ns (100 ps)
`timescale 1ns / 100ps


module testbench();

        // Processor signals
  logic  clk1, clk2;
  logic  reset;

        // Memory interface
        logic       MemWrite; // active high
        wire  [15:0] MemData;
  logic [7:0]  Adr;

        // Internal Signals and Combinational Logic
        // To be handled on PCB

        // SRAM interface
        logic ce; // chip enable, active low
        logic oe; // output enable, active low
        logic we; // write enable, active low

        assign ce = 1'b0;
        assign oe = MemWrite; // output enabled when not writing
        assign we = !MemWrite;

        // Instantiating the processor
  top dut(clk1, clk2, reset, MemWrite, Adr, MemData[14:8], MemData[7:0]);

        // Instantiating the SRAM
        sram mem(ce, oe, we, Adr, MemData);

        // Opening the testvector file and setting up memory
        // Pulsing reset to initialize the processor
        initial
        begin
                // NOTE: sram module automatically initializes memory
```

```
		// from memfile.dat
		clk2 = 0; reset = 1; #19; reset = 0;
	end

	// Creating the clock
	always
	begin
		clk1 <= 1; # 3; clk1 <= 0; # 2;
		clk2 <= 1; # 3; clk2 <= 0; # 2;
	end

	// Executing Instructions
	always @(negedge clk2)
	begin
		if(MemWrite) begin
			if(MemData === 16'bzzzzzzzz00101101) // && dut.dp.PC === 8'd32)
				$display("Test Successful!");
			else
				$display("Test Failed with MemData = %b", MemData);
				//$display("Test Failed on Line %d with MemData = %b",
dut.dp.PC, MemData);
				$stop;
		end
	end
endmodule
```

B.2 SRAM

```
// Generic SRAM for E190AT: VLSI Design Project
// Authors: Erik Meike, Dominique Mena, Caleb Norfleet, & Kaveh Pezeshki
// Created Spring 2019

module sram #(parameter ADDR_WIDTH=8,
                        parameter DATA_WIDTH=16)
                       (
                                input logic ce,    // Active low chip enable
                                input logic oe,    // Active low output enable
                                input logic we,    // Active low write enable

                                input logic [ADDR_WIDTH-1:0] adr,  // Address bus
                                inout wire  [DATA_WIDTH-1:0] data  // Data bus
                       );

        // internal variables
        reg  [DATA_WIDTH-1:0] data_out;   // caching requested data
        reg  [DATA_WIDTH-1:0] mem [(ADDR_WIDTH**2)-1:0]; // memory

        // Combinational logic: memory read tri-state
        assign data = (!ce && we && !oe) ? data_out : {(DATA_WIDTH){1'bz}};

        // Memory Read / Write Latches
        always_latch
                if (!ce && !we && oe) // if writing data
                        mem[adr] = data;
                else if (!ce && we && !oe) //if reading data
                        data_out = mem[adr];

        // Initial block to set up memory for testing
        initial begin
                $readmemb("memfile.dat", mem);
        end
endmodule
```

B.3 Testvectors

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	Program Memory															
2	Assuming base 10		Registers									Notes	Mem			
3	addr	Instruction	R0	R1	R2	R3	R4	R5	R6	R7						
4	0	setn R0, 40	40	x	x	x	x	x	x	x						
5	1	loadr R1, R0	40	5	x	x	x	x	x	x			5 already in addr 64			
6	2	copy R2, R1	40	5	5	x	x	x	x	x						
7	3	neg R2, R2	40	5	-5	x	x	x	x	x						
8	4	add R3, R0, R2	40	5	-5	35	x	x	x	x						
9	5	sub R4, R0, R2	40	5	-5	35	45	x	x	x						
10	6	jumpn 8	40	5	-5	35	45	x	x	x		sets R4 incorrectly if jump fails				
11	7	setn R4, 0	40	5	-5	35	0	x	x	x		We want R4 to remain 64 to pass test				
12	8	setn R5, 11	40	5	-5	35	45	11	x	x						
13	9	jumpr R5	40	5	-5	35	45	11	x	x		sets R4 incorrectly if jump fails				
14	10	setn R4, 0	40	5	-5	35	0	11	x	x						
15	11	setn R6, 0	40	5	-5	35	45	11	0	x						
16	12	jeqzn R6, 14	40	5	-5	35	45	11	0	x		should jump				
17	13	setn R4, 0	40	5	-5	35	0	11	0	x						
18	14	jeqzn R1, 30	40	5	-5	35	45	11	0	x		should not jump				
19	15	jeqzn R2, 30	40	5	-5	35	45	11	0	x		should not jump				
20	16	jnezn R1, 18	40	5	-5	35	45	11	0	x		should jump				
21	17	setn R4, 0	40	5	-5	35	0	11	0	x						
22	18	jnezn R2, 20	40	5	-5	35	45	11	0	x		should jump				
23	19	setn R4, 0	40	5	-5	35	0	11	0	x						
24	20	jnezn R6, 30	40	5	-5	35	45	11	0	x		should not jump				
25	21	jgtzn R1, 23	40	5	-5	35	45	11	0	x		should jump				
26	22	setn R4, 0	40	5	-5	35	0	11	0	x						
27	23	jgtzn R2, 30	40	5	-5	35	45	11	0	x		should not jump				
28	24	jgtzn R6, 30	40	5	-5	35	45	11	0	x		should not jump				
29	25	jltzn R2, 27	40	5	-5	35	45	11	0	x		should jump				
30	26	setn R4, 0	40	5	-5	35	0	11	0	x						
31	27	jltzn R1, 30	40	5	-5	35	45	11	0	x		should not jump				
32	28	jltzn R6, 30	40	5	-5	35	45	11	0	x		should not jump				
33	29	jumpn 31	40	5	-5	35	45	11	0	x						
34	30	setn R4, 0	40	5	-5	35	45	11	0	x		Fails test				
35	31	setn R7, 41	40	5	-5	35	45	11	0	41						
36	32	storer R4, R7	40	5	-5	35	45	11	0	41			Test succeedes if 45 written in addr 65			
37	Data Memory															
38	Addr	Stored Val														
39	40	5										set before test				
40	41	45, 0										45 if test passes, 0 if test fails				
41																

Figure 14: The HMMMM Test Program, with annotated register states after each instruction

C Custom Leaf Cell: Tri-State Buffer

C.1 Schematic

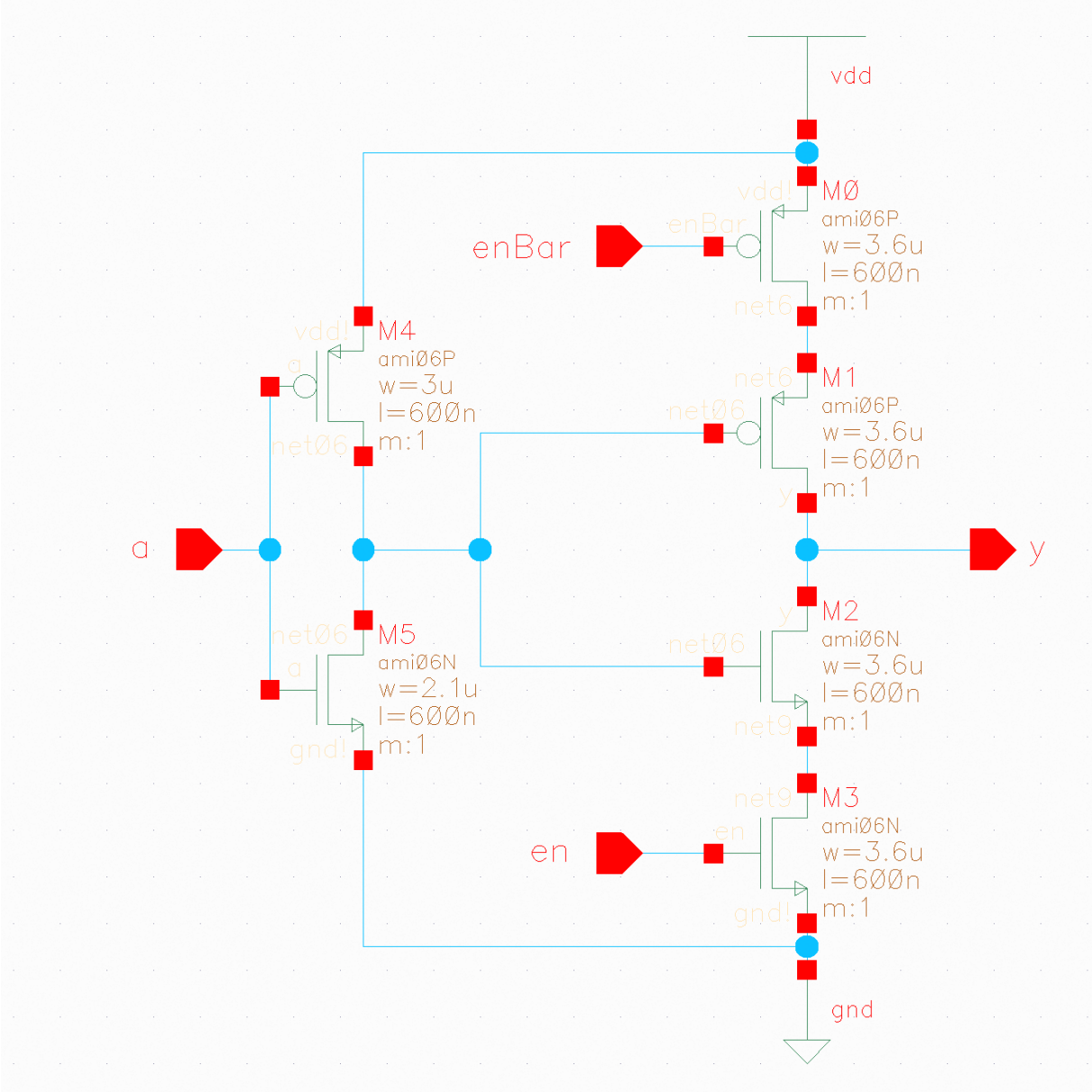

Figure 15: Schematic for the custom tri-state buffer leaf cell

C.2 Symbol

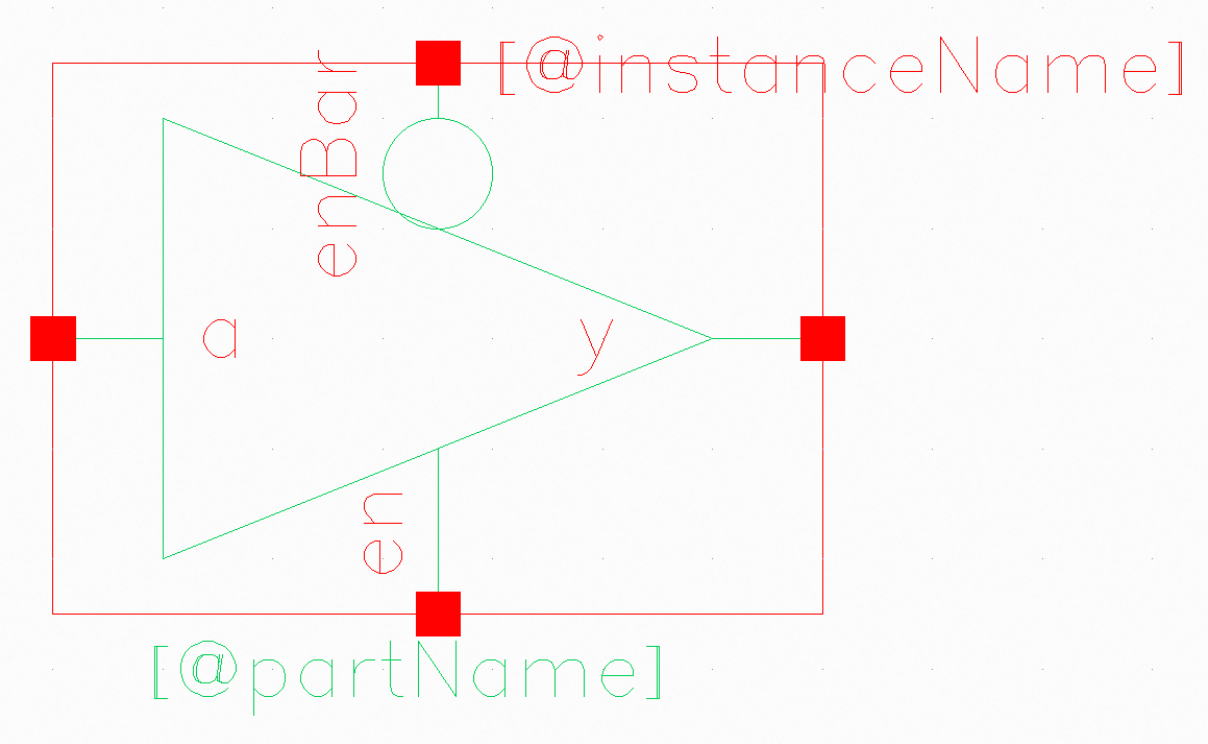

Figure 16: Symbol for the custom tri-state buffer leaf cell

C.3 Layout

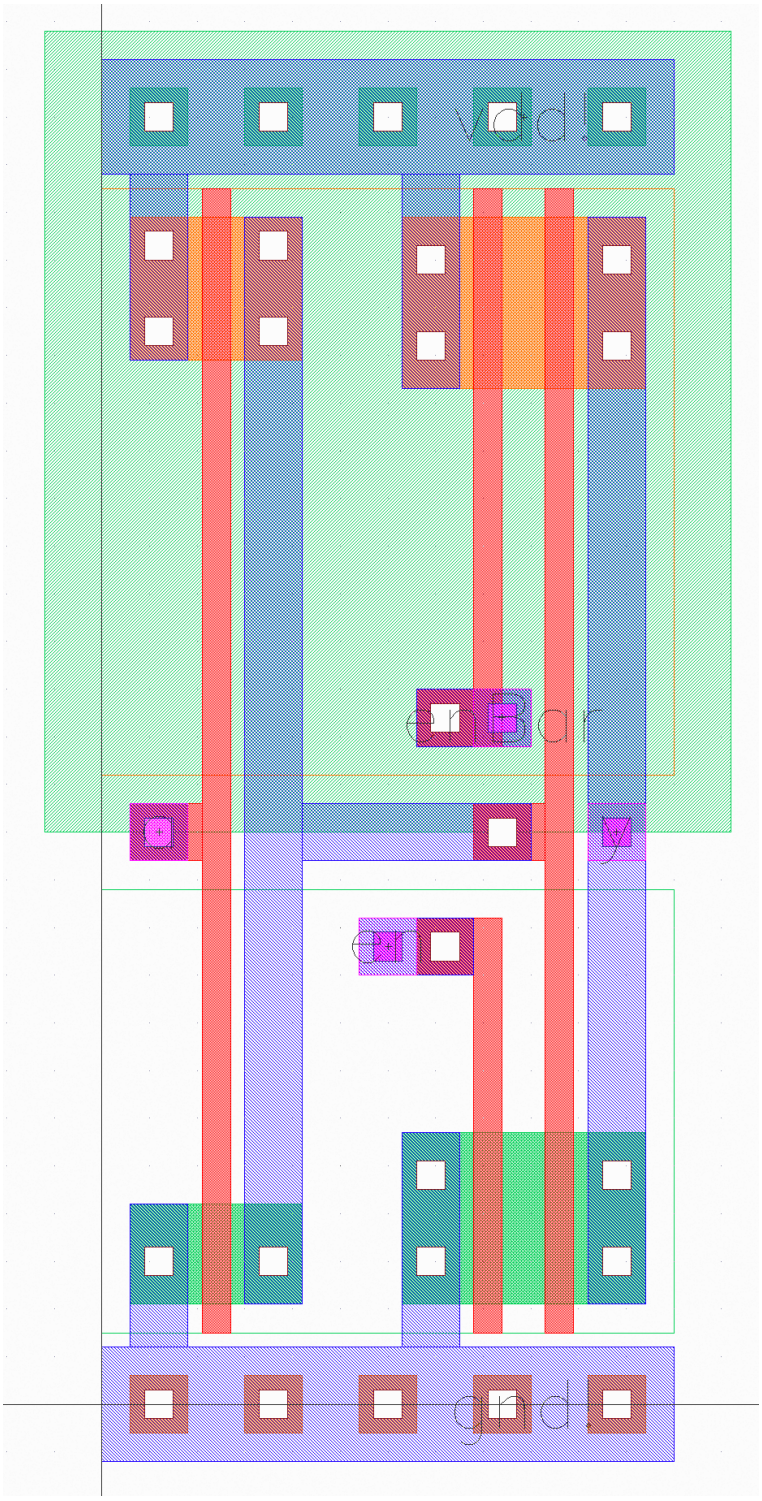

Figure 17: Layout for the custom tri-state buffer leaf cell

D Layouts

D.1 Controller

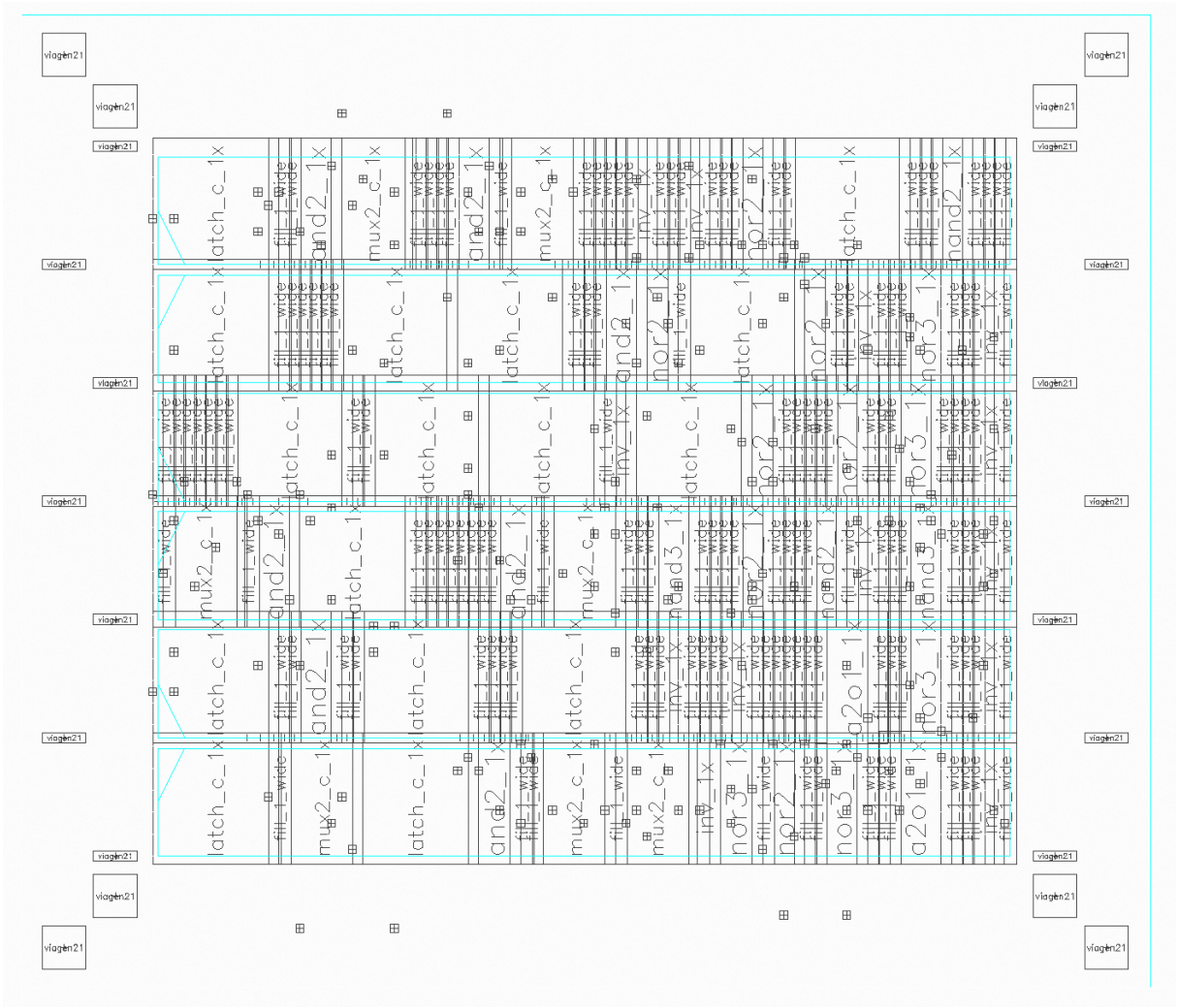

Figure 18: Layout image of the HMMM Controller. Each named block denotes a sub-cell

D.2 Datapath

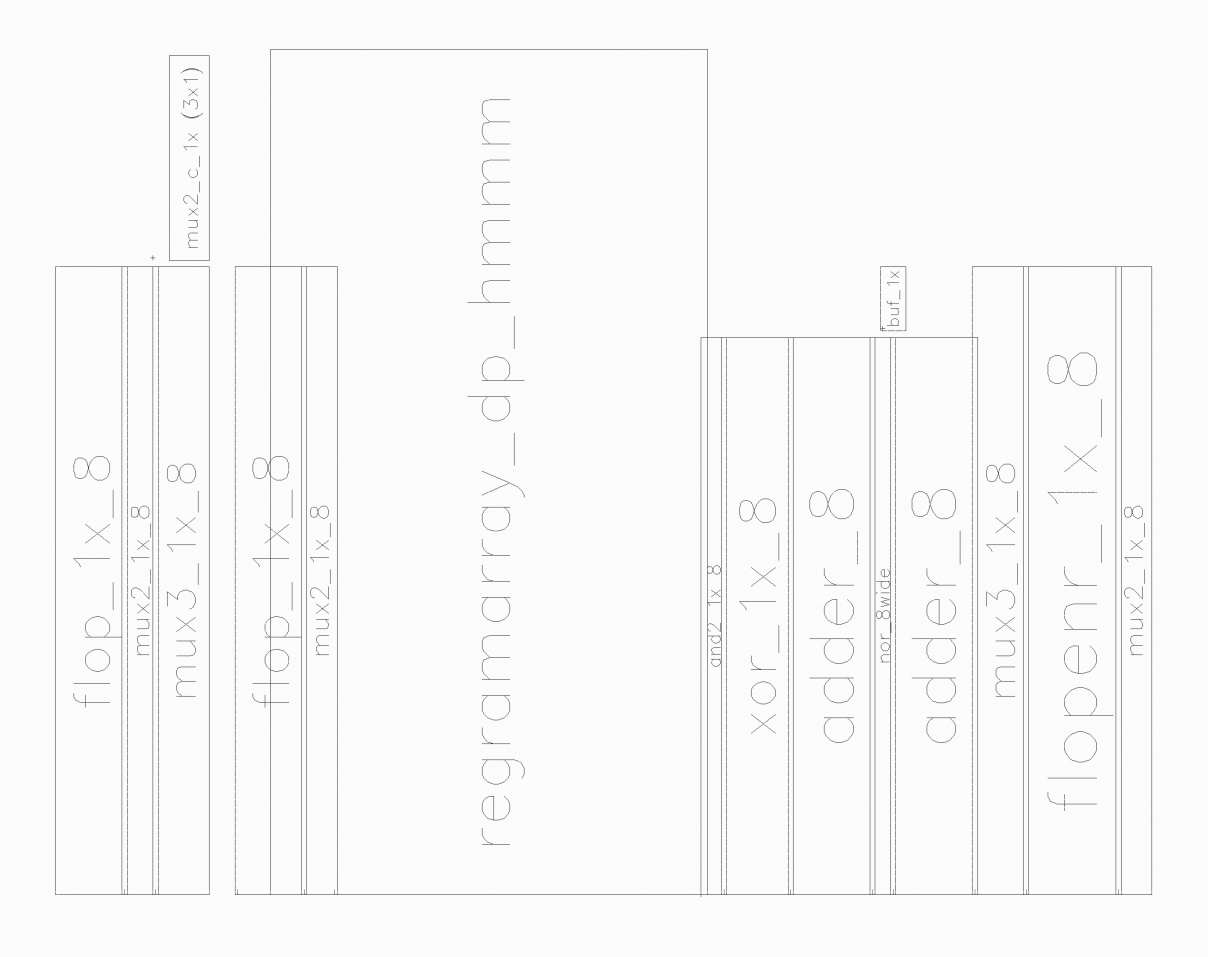

Figure 19: Layout image of the HMMM datapath. Each named block denotes a sub-cell