ECE 551 Semester Project - CVP14 Vector Co-Processor

Intermediate Report

Team “GET synth’D”
Ajay Sekar
David Gardner
Evan Thompson
Rakesh Ramananda

Table of Contents

Introduction

The CVP14 is a 16-bit vector operation coprocessor that shares a memory pool with a processor. It executes instructions that have been placed in memory by that processor. The CVP14 is intended to increase the speed of IEEE 754 half-precision floating point operations by offloading them from the processor. At this intermediate stage of the project, CVP14 implements 8 operations: Vector Load, Vector Store, Scalar Load Low, Scalar Load High, Scalar Store, Jump, Vector Add, and No Op. The Scalar Multiply and Vector Dot Product operations will be developed at a later date.

CVP14 Architecture

The CVP14 architecture consists of a vector register file, a scalar register file, and a unit which addresses the shared memory (see Appendix A for a block diagram). A state machine is used to control execution flow (see Appendix B).

State Machine Top-Level Architecture

The CVP14 coprocessor is implemented as a state machine which loads and executes one instruction at a time. The execution cycle begins with instruction fetching, which has two states. In the first, the shared memory is addressed with the program counter (PC). On the next clock, the instruction is read into the instruction register. Once the instruction has been fetched, a three-state execution process begins. In the first state, the instruction is decoded, the vector and scalar register files are addressed, and their read or write flags are set. Instructions which only take one clock cycle then proceed immediately to the done state, where the PC is incremented. Instructions which take more than one cycle enter an executing state, which proceeds to the done state when a done flag is detected. This done flag is sent by the units that implement the different instructions. This allows the execution cycle to vary in length depending on different instruction implementations, resulting in a CVP14 with a highly modular design, where instruction implementations can be independently optimized. If an overflow occurs during a vector operation, the implementing unit is responsible for setting an overflow flag. When this flag is detected, the CVP14 enters an overflow state in which the PC is set to hFFF0 and the current instruction is stored to scalar register seven. Then, the instruction at hFFF0 is fetched and execution resumes.

Two-phase Clock

The shared system memory uses a two-phase clock. Within CVP14, the scalar and vector registers implement the same two-phase clocking system. Flags and data inputs are read on clock one, while addresses and data outputs are read on clock two.

Shared Memory Addressing

CVP14 has three internal modules which work to address the shared system memory. The address unit module controls what value is at the shared memory address. It has three settings: keep the address the same, set the address to the PC, and set the address to a 16-bit base plus a 6-bit instruction immediate plus a 4-bit offset. The instruction immediate and offset are sign-extended by the address unit module as well. The 4-bit offset is controlled by the offset unit module and is used for vector load and store operations. When the increment flag of the offset unit module is held high, the 4-bit offset changes from 0 to 1, 2, …, 15, and then back to 0 on each subsequent clock cycle. This behavior enables a serial load or store of vector elements between the shared memory and the CVP14 vector register file. The final module involved in shared memory addressing is the PC unit module. This unit contains an adder that becomes enabled when an update PC flag is set. It then adds either one, or a jump offset, to the PC depending on whether the current instruction is a jump instruction. The PC unit module also has a reset input that is used to set the PC to a starting address of h0000.

Scalar register module

The scalar register module is internal to CVP14 and contains eight 16-bit scalar registers. Each register is assigned an address, starting at zero and proceeding sequentially to seven. The scalar register module has the following four interface commands: read, write, write high, and write low. Write high places an 8-bit immediate into upper 8-bits of the chosen scalar register, whereas write low places the immediate into lower 8 bits. The clock cycles that are used in CVP14 are also used in this module. During clock cycle one, the correct command is executed. If no command is supplied (because the flags are all low), all scalar register retain their value. During clock cycle two the chosen scalar register address is updated.

Vector register module

The vector register module is also internal to CVP14 and contains eight registers that store 16-dimensional 16-bit vectors in IEEE 754 half precision floating point number form. Each register is assigned an address, starting at zero and proceeding sequentially to seven. The individual vector registers can be accessed either in their entirety (in parallel), or one 16-bit element at a time for 16 cycles (serially). This feature gives flexibility to the vector register, allowing it to be used in which ever format is more optimal. In addition to this feature, the vector register module is capable of accessing two individual vector registers at the same time. It can take in two vector addresses and produce two vector outputs, in either parallel or serial form. This optimizes operations that require two source vector registers.

The serial or parallel access to the vector register gives it four interface commands: read parallel, write parallel, read serial, and write serial. Based on the given command, the specified

vector register will have its contents accessed 256-bits at a time, or 16-bits at a time for 16 cycles. If no command is given then the contents of all vector registers are left unchanged. Similarly to the scalar register module, the clock cycles from CVP14 are taken into the vector register module. The appropriate command is executed on clock cycle one, and the chosen vector addresses are updated on clock cycle two.

Vector adder

VADD module is the vector addition unit of the WISC- CVP14 processor.The module accepts 16 dimensional 16 bit vectors and produces a 256 bit output. The module takes in a start flag to initiate the floating point addition, and on completion, it returns a done flag to the top module. Currently, we have 16 floating points adders to perform the vector addition, and we look forward to reducing the number of adders implemented to get an optimized result taking in area and time constraints. Representing the mantissa in the normalized form is taken care at all stages of floating point addition. Adder also checks for an overflow condition during addition and reports its to the processor as a flag register. the top level module checks this flag regularly and changes the state of processor as described in the specification.

Testbench

The testbench for CVP14 instantiates a shared system DRAM and loads it with a series of instructions and vectors. These instructions test every part of CVP14 functionality, including corner cases such as overflow.

Post-Synthesis Performance

CVP14 is fully functional post synthesis, as shown by the shared memory dump in Appendix C. This shared memory contains a test of every instruction, including a vector add that causes overflow.

Individual Contributions

The tasks that needed to be completed for the intermediate project milestone were divided amongst the group. Evan and David were assigned the top level module design. This consisted of creating the CVP14, scalar register, and vector register modules. The operations contained inside CVP14, such as SLL, SLH, NOP, J, and SST, were also implemented. Evan spearheaded the overall state machine design for CVP14 and added additional modules that helped keep CVP14 abstract. These modules were the addressing unit, PC unit, and offset unit. The block diagram of CVP14 and state machine diagram were created by David after the design was finalized. David and Evan worked together on creating a test bench for every module.

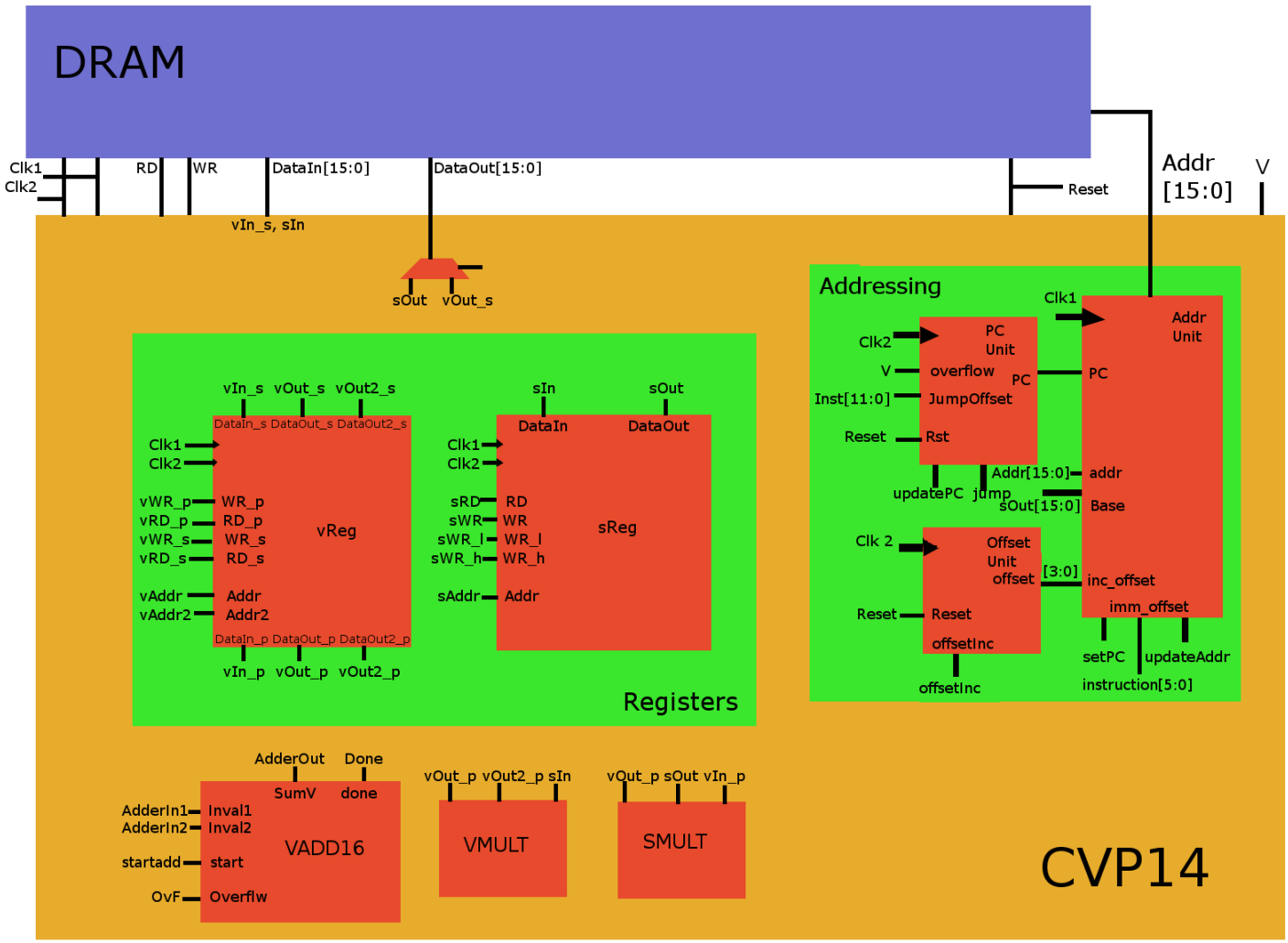
The remaining modules to be implemented were VLD, VST and VADD. These operations were assigned to Rakesh and Ajay. Rakesh worked on designing floating point adder unit. Initially he worked on designing a 16-bit carry look ahead adder, ripple carry adder and adder using behavioural verilog. He compared area and timing constraints of all 3 designs and concluded that adder built using behavioral and carry look ahead had same area and timing. The vector ADD module now has a rounding mechanism and overflow detection unit along with the capability to handle denormalized numbers. Ajay worked on the VADD module and got a functional block which was not working during post synthesis. Rakesh and Ajary worked together to create a functioning VADD.

Due to the high level of difficulty in implementing VADD, VST and VLD were then assigned to David and Evan to complete.

Future Plans

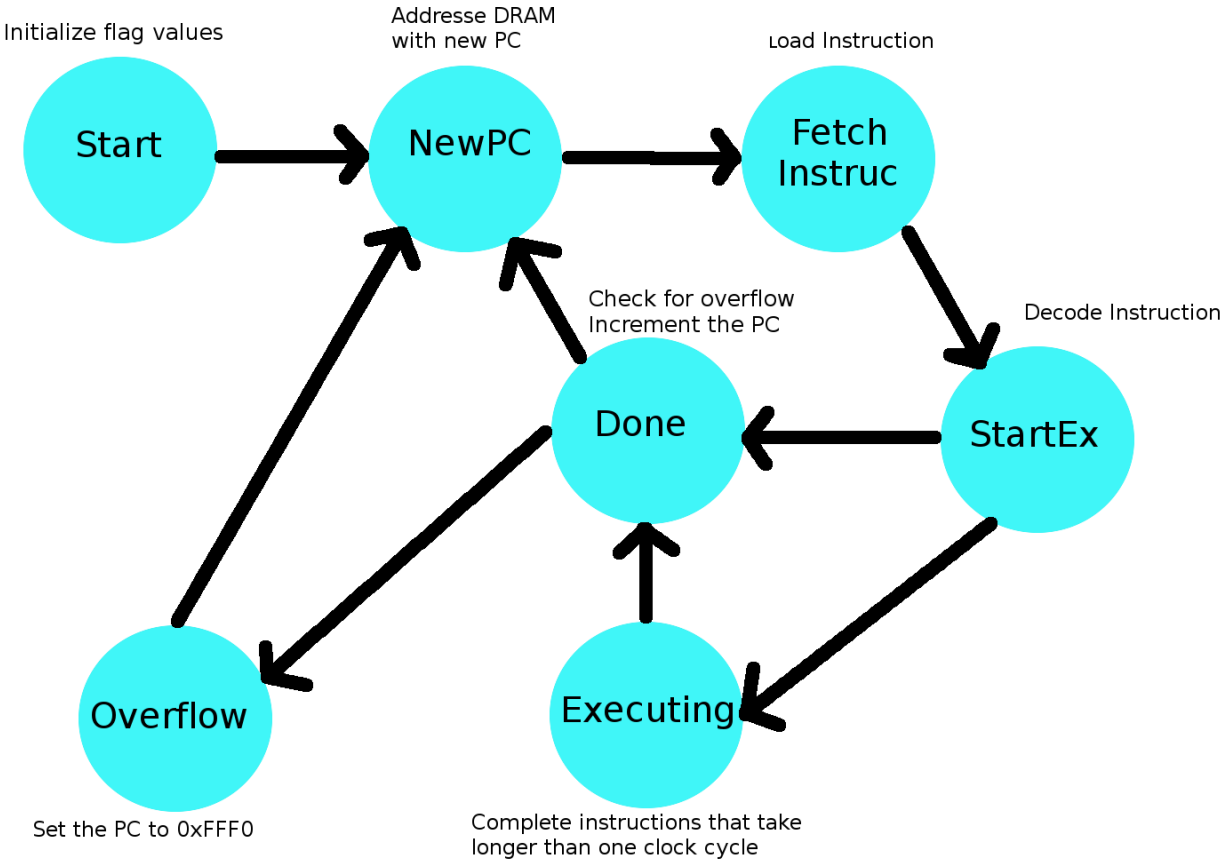
The future development of this project consists of completing the SMUL and VDOT operations, in addition to creating multiple designs for the project. David and Evan will complete the VDOT operation. Rakesh and Ajay will complete the SMUL operation. Once all operations are successfully implemented, design optimizations and varying designs will be completed. The synthesis tool will be run under varying constraints to optimize different things such as timing, area and performance. Varying designs will include making VADD work serially instead of in parallel and reducing the area of the vector register by removing the parallel capability.

Appendix A


DRAM
Clk1
Clk2
RD
WR
DataIn[15:0]
DataOut[15:0]
vIn_s, sIn
sOut
vOut_s
Reset
Addr
[15:0]
V
Addressing
Clk1
Clk2
PC
Unit
V
overflow
PC
Inst[11:0]
JumpOffset
Reset
Rst
updatePC
jump
Addr[15:0]
sOut[15:0]
addr
Base
Addr
Unit
PC
Clk 2
Offset
Unit
offset
[3:0]
inc_offset
imm_offset
Reset
Reset
offsetInc
offsetInc
setPC
updateAddr
instruction[5:0]
vIn_s
vOut_s
vOut2_s
DataIn_s DataOut_s DataOut2_s
Clk1
Clk2
vWR_p
vRD_p
vWR_s
vRD_s
WR_p
RD_p
WR_s
RD_s
vReg
vAddr
vAddr2
Addr
Addr2
DataIn_p DataOut_p DataOut2_p
vIn_p
vOut_p
vOut2_p
sIn
sOut
DataIn
DataOut
Clk1
Clk2
sRD
sWR
sWR_l
sWR_h
RD
WR
WR_l
WR_h
sReg
sAddr
Addr
Registers
AdderOut
Done
SumV
done
AdderIn1
AdderIn2
Inval1
Inval2
VADD16
startadd
start
OvF
Overflw
vOut_p vOut2_p sIn
VMULT
vOut_p sOut vIn_p
SMULT
CVP14

Appendix B

State Machine Diagram



Appendix C

This is the dump of memory after a test bench run on the post synthesis CVP14 module. It is commented to represent the passed in operations and stored results that show the operations were completed successfully.

```
1 // memory data file
//Memory address 0 starts at line 4
4 1111000000000000      //NOOP
5 0110000011101111      //SLL S0 0xEF
6 0110111000010010      //SLL S7 0x12
7 0111000001110000      //SLH S0 0x70
8 0111001010101011      //SLH S1 0xab
9 0110110001010000      //SLL S6 0x50
10 0111110000000000     //SLH S6 0x00
11 1000000000000111     //J 0000_0000_0111 Jump to line 18
12 xxxxxxxxxxxxxxxx
17 xxxxxxxxxxxxxxxx     //The PC jumps to line 18 to continue execution
18 0011000110000000     //SST S0 S6 000000
19 0110110010100000     //SLL S6 0xa0
20 0100000110000000     //VLD V0 S6 000000
21 0111111001110000     //SLH S7 0x70
22 0110011011110000     //SLL S3 0xf0
23 0111011000000000     //SLH S3 0x00
24 0101000011000000     //VST V0 S3 000000
25 0110000011111111     //SLL S0 0xff
26 0110000000000000     //SLL S0 0x00
27 0111000000001100     //SLH S0 0x0C
28 0100001000000000     //VLD V1 S0 000000
29 0111110000000000     //SLH S6 0x00
30 0110110010100000     //SLL S6 0xa0
31 0100000110000000     //VLD V0 S6 000000
32 0000010001000000     //VADD V2 V1 V0, will cause overflow skipping next ops
33 0110011000000000     //SLL S3 0x00
34 0111011000001010     //SLH S3 0x0A
35 0101010011000000     //VST V2 S3 000000
36 xxxxxxxxxxxxxxxx
```

Scalar Registers at this point:
S0 = 0x70EF, after line 25 = 0x0c00
S1 = 0xabXX
S3 = 0x00f0, after line 32 = 0x00f0
S6 = 0x0050, after line 18 = 0x00a0
S7 = 0x7012

```
83 xxxxxxxxxxxxxxxx
84 0111000011101111    // = 0x70EF, Represents the SST S0 S6 at line 18
// Represents the instruction that caused overflow, see line 65526
85 0000010001000000
86 xxxxxxxxxxxxxxxx

163 xxxxxxxxxxxxxxxx  //Vector to load into V0, see line 20
164 0100100010010000
165 0101001100111010
166 0001011001100100
167 0011100000000000
168 0000111100000000
169 0011111100000000
170 0011111100000000
171 0011100000000000
172 0111111100000000
173 0011110100000000
174 0111101000000000
175 0111101000000000
176 0000000000000001
177 0000000000000001
178 0000000000000001
179 0000000000000000
180 xxxxxxxxxxxxxxxx //End V0

243 xxxxxxxxxxxxxxxx  //V0 stored into memory, see line 24
244 0100100010010000
245 0101001100111010
246 0001011001100100
247 0011100000000000
248 0000111100000000
249 0011111100000000
```

250 0011111100000000
251 0011100000000000
252 0111111100000000
253 0011110100000000
254 0111101000000000
255 0111101000000000
256 0000000000000001
257 0000000000000001
258 0000000000000001
259 0000000000000000
260 xxxxxxxxxxxxxxxx// End V0

//Where V2 would have been stored if no overflow occurred (VST was skipped)
2564 xxxxxxxxxxxxxxxx
2565 xxxxxxxxxxxxxxxx
2566 xxxxxxxxxxxxxxxx
2567 xxxxxxxxxxxxxxxx
2568 xxxxxxxxxxxxxxxx
2569 xxxxxxxxxxxxxxxx
2570 xxxxxxxxxxxxxxxx
2571 xxxxxxxxxxxxxxxx
2072 xxxxxxxxxxxxxxxx
2073 xxxxxxxxxxxxxxxx
2074 xxxxxxxxxxxxxxxx
2075 xxxxxxxxxxxxxxxx
2076 xxxxxxxxxxxxxxxx
2077 xxxxxxxxxxxxxxxx
2078 xxxxxxxxxxxxxxxx
2079 xxxxxxxxxxxxxxxx //End what would have been V2

3075 xxxxxxxxxxxxxxxx //Load into V1, see line 28
3076 0100000101010101
3077 0101001100111010
3078 1000111100110000
3079 1011010000000000
3080 1001011110000000
3081 1011110100000000
3082 0010101110010001
3083 0000000111100101

3084 0000000111100101
3085 0011010111100101
3086 0111101011110010
3087 0111011011110010
3088 0000000000000001
3089 0000000000000001
3090 0000000000000001
3091 0000000000000000
3092 xxxxxxxxxxxxxxxx //End V1

65523 xxxxxxxxxxxxxxxx //This is the overflow code section
65524 0111101000000000 //SLH S5 0x00
65525 0110101001010001 //SLI S5 0x51
//This stores the instruction that caused the overflow to line 85
65526 0011111101000000 //SST S7 S5 000000
65527 xxxxxxxxxxxxxxxx

//Spencer's testbench memory dump

```
 0 1111000000000000      //NO OP
 1 1111000000000000      //NO OP
 2 0110000000000000      //SLL S0 0x00
 3 0111000000000001      //SLH S0 0x01
 4 0100000000000000      //VLD V0 S0 0x00
 5 1111000000000000      //NO OP
 6 0000000000000000      //VADD V0 V0 V0
 7 0101000000000000      //VST V0 S0 0x00
 8 0110000011111111      //SLL S0 0xff
 9 0111000011111111      //SLH S0 0xff
10 0100000000000000      //VLD V0 S0 0x00
```

Scalar Registers: S0 = 0x0100, after line 7 = 0xffff

//Result of VADD at line 6, contents of V0

```
256 1001110100111001    //0x9d39, repeated throughout V0
257 1001110100111001
258 1001110100111001
259 1001110100111001
260 1001110100111001
261 1001110100111001
262 1001110100111001
263 1001110100111001
264 1001110100111001
265 1001110100111001
266 1001110100111001
267 1001110100111001
268 1001110100111001
269 1001110100111001
270 1001110100111001
271 1001110100111001    //End contents of V0
272 1001100100111001    //Extra values in memory : 0x9339
273 1001100100111001
274 1001100100111001
275 1001100100111001
276 1001100100111001
2771001100100111001
278 1001100100111001
279 1001100100111001
```