ESE 345 : Computer Architecture Project

Daniel Cao (109265704)

Ricky Zhang (109311456)

Goals:

The main goal of our project is to use VHDL , a hardware description language in order to design the behavior of a triple stage pipelined multimedia Cell-Lite unit with reduced multimedia functions just like how it was done in the Sony Cell SPU architecture. The use of cad tools such as a VHDL/Verilog simulator was used to test our results. In our case, we used Aldec Active HDL simulator to write and test our code in VHDL.

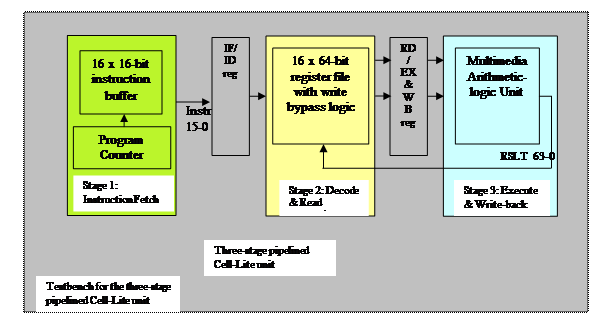


Figure 1: Diagram of our three stage pipeline unit

Procedures:

In order to implement the Three-stage pipelined Cell-lite unit, we first implemented the ALU. The ALU consisted of two major sub-modules, each with different functions. These modules are the arithmetic module which consist of the adding and subtractions functions and the logic and shift module which consist of all the logic and shift operations. In order to do the arithmetic module, we needed to create a 16-bit/32-bit arithmetic module using a triple level carry look ahead system. We tested our carry-look ahead system using the simulation in AHDL. We input values into our registers and using a mux to select whether it was addition or subtraction, we tested our CLA.

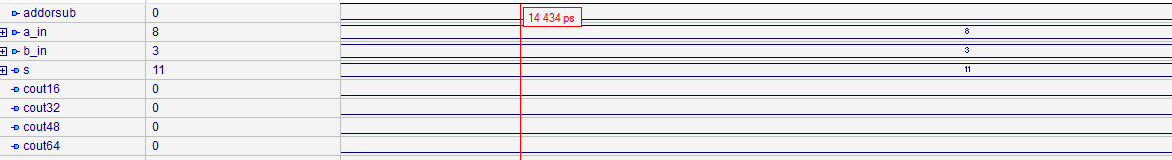


Figure 2: addition

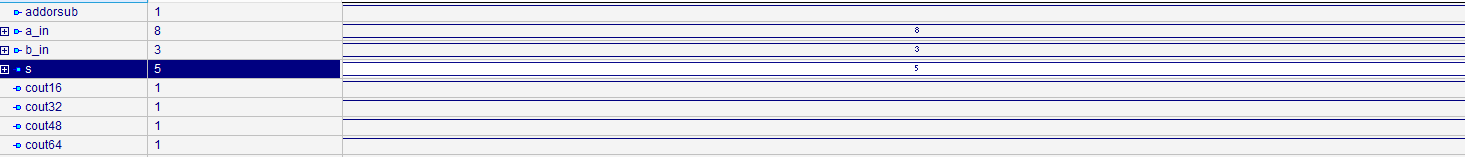


Figure 3: subtraction

We next implemented the register file. The register file consist of 16 64-bit registers and three components that are in charge of decoding, reading and writing. On any cycle, the register file should be able to do 2 reads and 1 write. Ideally, in each cycle, both the 64-bit register is read and one 64 –bit register value is written. Furthermore, a forwarding system should be implemented so that when a read and write function happens to the same register, it will return the new value read and not the old value. The register file was implemented using a behavioral model in VHDL. In our decode component, the binary values are split and decoded. Within our read component, data is stored and written back on the falling edge of the clock cycle. Instructions are read within the test bench and decoded. The values of our op codes can be seen below relating to their specific instruction.

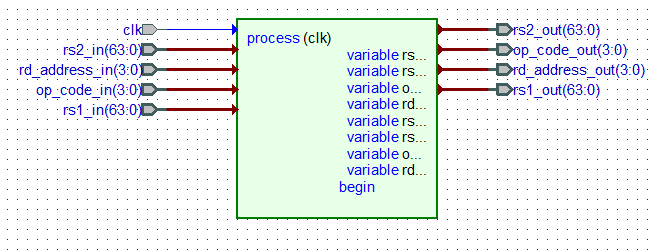


Figure 4 :Code to graphics of our register file

|  |  |
| --- | --- |
| **OPCODE 3-0** | **Description of Instruction Opcode** |
| 0000 | **nop** |
| 0001 | **lv**: Load a 8-bit immediate value from the [11:4] instruction field into **the every byte**of register ***rd.****(This is done by bypassing the ALU’s arithmetic and logical/shift modules. The****lv****'execution" takes the same amount of time as all other operations.)* |
| 0010 | **and**: bitwise ***logical and*** of the contents of registers ***rs1*** and ***rs2*** |
| 0011 | **or**: bitwise ***logical or*** of the contents of registers ***rs1*** and ***rs2*** |
| 0100 | **cnth**:*count ones in halfwords*: the number of 1s in each of the four halfword-slots in register ***rs1*** is computed. If the halfword slot in register ***rs1***is zero, the result is 0. Each of the results is placed into corresponding 16-bit slot in register ***rd***. ***(Comments: 4 separate 16-bit halfword values  in each 64-bit register)*** |
| 0101 | **clz**:*count leading zeroes in words*: for each of the two 32-bit word slots in register ***rs1*** the number of zero bits to the left of the first “1” is computed. If the word slot in register ***rs1***is zero, the result is 32. The two results are placed into the corresponding 32-bit word slots in register ***rd***. ***(Comments: 2 separate 32-bit values in each 64-bit register)*** |
| 0110 | **rot:***rotate right:*the contents of register ***rs1*** are rotated to the right according to the count in the 6 least significant bits (5 to 0) of the contents of register ***rs2***. The result is placed in register ***rd***. If the count is zero, the contents of register ***rs1*** are copied unchanged into register ***rd***. Bits rotated out of the right end of the 64-bit contents of register ***rs1*** are rotated in at the left end. |
| 0111 | **shlhi**: *shift left halfword immediate*: packed 16-bit halfword shift left logical of the contents of register ***rs1*** by the 4-bit immediate value of instruction field ***rs2***. Each of the results is placed into the corresponding 16-bit slot in register ***rd***. ***(Comments: 4 separate 16-bit values in each 64-bit register)*** |
| 1000 | **a**: *add word*: packed 32-bit unsigned add of the contents of registers ***rs1*** and ***rs2 (Comments: 2 separate 32-bit values in each 64-bit register)*** |
| 1001 | **sfw**: *subtract from word*: (packed) 32-bit unsigned subtract of the contents of registers ***rs1*** and ***rs2 (Comments: 2 separate 32-bit values in each 64-bit register)*** |
| 1010 | **ah**: *add halfword* : (packed) (16-bit) halfword unsigned add of the contents of registers ***rs1*** and ***rs2*** ***(Comments: 4 separate 16-bit values in each 64-bit register)*** |
| 1011 | **sfh**: *subtract from halfword*: (packed) (16-bit) halfword unsigned subtract of the contents of registers ***rs1*** and ***rs2*** |
| 1100 | **ahs**: *add halfword* *saturated*: (packed) (16-bit) halfword unsigned add **with saturation** of the contents of registers ***rs1*** and ***rs2*** |
| 1101 | **sfhs**: *subtract from halfword* *saturated*: (packed) (16-bit) unsigned subtract **with saturation** of the contents of registers ***rs1*** and ***rs2*** |
| 1110 | **mpyu**:  *multiply unsigned*: the 16 rightmost bits of each of the two 32-bit slots in registers ***rs1*** are multiplied by the 16 rightmost bits of the corresponding 32-bit slots in register ***rs2***, treating both operands as unsigned. The two 32-bit products are placed into the corresponding slots of register ***rd***. ***(Comments: 2 separate 32-bit values in each 64-bit register)*** |
| 1111 | **absdb**:  *absolute difference of bytes*: the contents of each of the eight byte slots in register***rs2*** is subtracted from the contents of the corresponding byte slot in register ***rs1***. The absolute value of each of the results is placed into the corresponding byte slot in register ***rd***. ***(Comments: 8 separate 8-bit values in each 64-bit register)*** |

Figure 5: Op-code instructions

The third module that needed to be made was the instruction buffer. Within the instruction buffer, the contents of the test bench would be loaded on the positive edge of the clock. Furthermore, a program counter was implemented in the module so that it would count each time an instruction is implemented.

The opcodes that we implanted corresponding to the simulation at the bottom is here.

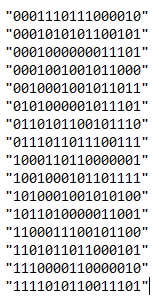


Figure 6: 16Bit instructions in our test bench

The instructions can be decoded using the instruction formats and opcode description.

|  |
| --- |
|  |
|  |
|  |  |

The last step to complete our pipelined system was to create all the pipeline registers and put the system together. The pipeline system will have three stages, IF, ID, and EXE stages.

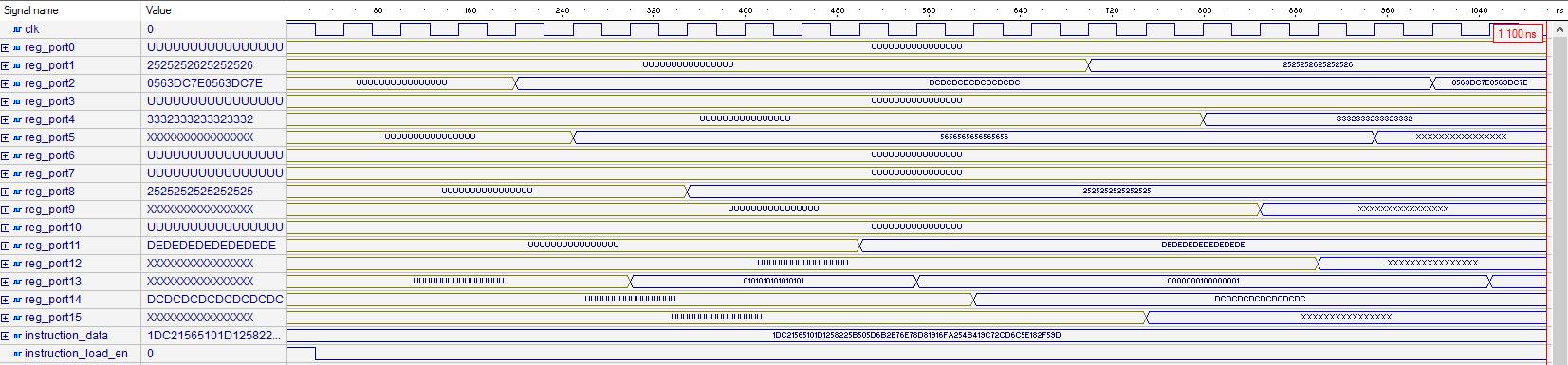


Figure 7: Simulation of our inputted opcodes

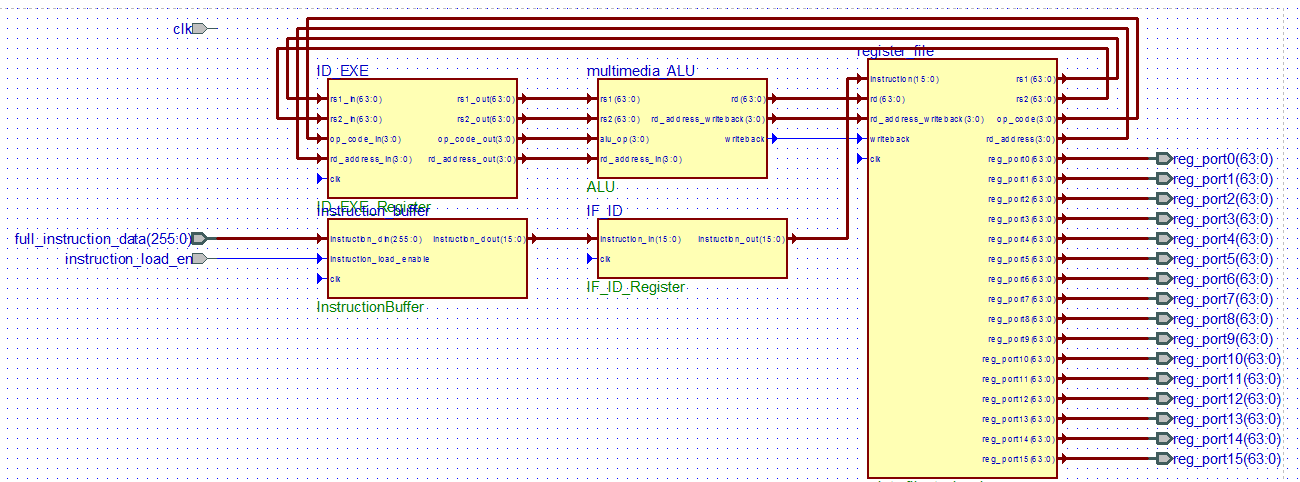


Figure 8: Code to graphics of our final pipeline system.

Our Cell-Lite Unit is a three stage pipelined system. Within IF/ID stage, the first stage of the processor, the opcodes are loaded in and decoded. The 16 bit instructions are fetched from the test bench and are send to the register file at the rising edge of the clock. In the ID/RD stage, the instruction will be determined by the opcode.