ESE 345 Fall 2021 Final Project

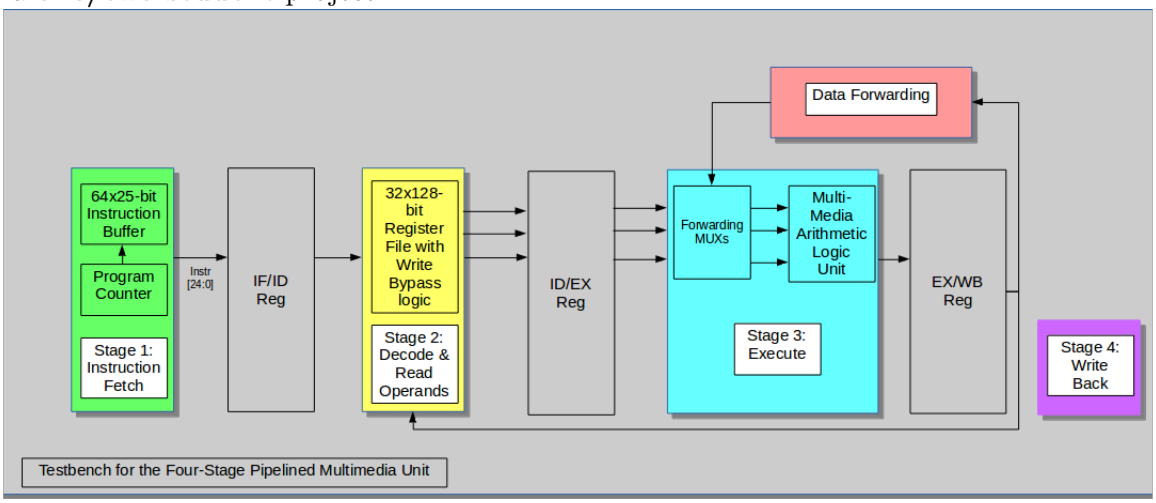
Pipelined multimedia unit design with the VHDL/Verilog hardware description language

1 Introduction

Purpose: To learn a use of VHDL/Verilog hardware description language and modern CAD tools for the structural and behavioral design of a four-stage pipelined multimedia unit with a reduced set of multimedia instructions similar to those in the Sony Cell SPU and Intel SSE architectures.

CAD Tools: Mentor Graphics Modelsim at the Undergraduate CAD Lab (room 281 Light Eng. Bldg.) or any other VHDL/Verilog simulator (e.g. Aldec, Vivado).

It is a **one/two-student** project



2 Procedure

1. It is suggested to read **Chapter 3.6** on subword parallelism and, if necessary, the Intel MMX and Sony Cell SPU[1] papers below and understand the original concept of multimedia processing introduced as MMX architecture for Intel processors in the 1990s.
2. **Refresh your knowledge** of VHDL/Verilog in the HDL design of digital circuits by reading **Chapter 4.13**.
3. **Develop** a detailed block diagram and the HDL model of the four-stage multimedia unit and its modules.
4. **Verify** individual modules of your design with their testbenches before instantiating them in higher order modules. Verify the final model with a testbench module and generate file **Results** showing the status of each stage of the unit during execution.

3 Requirements

The complete 4-stage pipelined design is to be developed in a structural/RTL manner with several modules operating simultaneously. Each module represents a pipelined stage with its interstage register. The major units inside those stages modules are described below.

1. **Multimedia ALU**
   Takes up to three inputs from the Register File, and calculates the result based on the current instruction to be performed.
   The ALU must be implemented as **behavioral model in VHDL or continuous assignment (dataflow models in Verilog)**.

2. **Register File**
   The register file has 32 128-bit registers. On any cycle, there can be 3 reads and 1 write. When executing instructions, each cycle two/three 128-bit register values are read, and one 128-bit result can be written if a write signal is valid. This ***register write*** signal must be explicitly declared so it can be checked during simulation and demonstration of your design.
   The register module must be implemented as **a behavioral model in VHDL (dataflow/RTL model in Verilog)**.

3. **Instruction Buffer**
   The instruction buffer can store 64 25-bit instructions. The contents of the buffer should be loaded by the testbench instructions from a test file at the start of simulation. On each cycle, the instruction specified by the Program Counter (PC) is fetched, and the value of PC is incremented by 1.
   The Instruction Buffer module must be implemented as **a behavioral model in VHDL (dataflow/RTL model in Verilog)**.

4. **Forwarding Unit**
   Every instruction must use the **most recent value** of a register, even if this value has not yet been written to the Register File. Be mindful of the ordering of instructions; the most recent value should be used, in the event of two consecutive writes to a register, followed by a read from that same register. Your processor should never stall in the event of hazards.
   Take extra care of which instructions require forwarding, and which ones do not. Namely, NOP (wr_enabled signal low) and Immediate fields (doesn't contain a register source). Only valid data and source/destination registers should be considered for forwarding.

5. **Four-Stage Pipelined Multimedia Unit**
   Clock edge-sensitive pipeline registers separate the IF, ID, EXE, and WB stages. Data should be written to the Register File after the WB Stage.
   All instructions (including **li**) take **four** cycles to complete. This pipeline must be implemented as a structural model with modules for each corresponding pipeline stages and their interstage registers. Four instructions can be at different stages of the pipeline at every cycle.

6. **Testbench** This module loads the instruction buffer using data loaded from a file, begins simulation, and upon completion, **compares the contents of the register file** to a file containing the expected results. This expected results file does not need to be auto-generated. Instead, this can be manually entered when designing a test program.
   This must be implemented as a **behavioral model**.

7. **Assembler** This is a separate program written in any language your team prefers (i.e. Java, C++, Python). Its purpose is to convert an assembly file to the binary format for the Instruction Buffer. This assembler does not need to be robust, and can assume very specific syntax rules that you as a team decide.

8. **Results File** This file must show the status of the pipeline for each cycle during program execution. It should include the opcodes, input operand, and results of the execution of instructions, as well as all relevant control signals and forwarding information. This should be carried out by your testbench.

4 Instruction Formats and Opcode Description

4.1 Load Immediate

24	23 21	20 5	4 0
0	Load Index	Immediate	rd

li: Load a 16-bit Immediate value from the [20:5] instruction field into the 16-bit field specified by the li field [23:21] of the 128-bit register rd.

4.2 Multiply-Add and Multiply-Subtract R4-Instruction Format

24	23	22	21	20	19 15	14 10	9 5	4 0
1	0	Long/Int	Subtract/Add	High/Low	rs3	rs2	rs1	rd

Signed operations are performed with **saturated** rounding that takes the result, and sets a floor and ceiling corresponding to the max range for that data size. This means that instead of over/underflow wrapping, the max/min values are used.

Size (Num Bits)	Min	Max
Long (64)	-2^{63}	$+2^{63}-1$
Int (32)	-2^{31}	$+2^{31}-1$

The tables below show the description for each operation:

LI/SA/HL [22:20]	Description of Instruction Code
000	**Signed Integer Multiply-Add Low with Saturation**: Multiply low 16-bit-fields of each 32-bit field of registers **rs3** and **rs2**, then add 32-bit products to 32-bit fields of register **rs1**, and save result in register **rd**
001	**Signed Integer Multiply-Add High with Saturation**: Multiply high 16-bit-fields of each 32-bit field of registers **rs3** and **rs2**, then add 32-bit products to 32-bit fields of register **rs1**, and save result in register **rd**
010	**Signed Integer Multiply-Subtract Low with Saturation**: Multiply low 16-bit-fields of each 32-bit field of registers **rs3** and **rs2**, then subtract 32-bit products from 32-bit fields of register **rs1**, and save result in register **rd**
011	**Signed Integer Multiply-Subtract High with Saturation**: Multiply high 16-bit-fields of each 32-bit field of registers **rs3** and **rs2**, then subtract 32-bit products from 32-bit fields of register **rs1**, and save result in register **rd**
100	**Signed Long Integer Multiply-Add Low with Saturation**: Multiply low 32-bit-fields of each 64-bit field of registers **rs3** and **rs2**, then add 64-bit products to 64-bit fields of register **rs1**, and save result in register **rd**
101	**Signed Long Integer Multiply-Add High with Saturation**: Multiply high 32-bit-fields of each 64-bit field of registers **rs3** and **rs2**, then add 64-bit products to 64-bit fields of register **rs1**, and save result in register **rd**
110	**Signed Long Integer Multiply-Subtract Low with Saturation**: Multiply low 32-bit-fields of each 64-bit field of registers **rs3** and **rs2**, then subtract 64-bit products from 64-bit fields of register **rs1**, and save result in register **rd**
111	**Signed Long Integer Multiply-Subtract High with Saturation**: Multiply high 32-bit-fields of each 64-bit field of registers **rs3** and **rs2**, then subtract 64-bit products from 64-bit fields of register **rs1**, and save result in register **rd**

4.3 R3-Instruction Format

24	23	22 15	14 10	9 5	4 0
1	1	opcode	rs2	rs1	rd

In the table below, 16-bit signed integer add (**ahs**) and subtract (**sfhs**) operations are performed with **saturate to signed word** rounding that takes a 16-bit signed integer X, and converts it to -32768 if it's less than -32768, to +32767 if it's greater than 32767, and leaves it unchanged otherwise.

Opcode [22:15]	Description of Instruction Opcode
xxxx0000	**NOP**
xxxx0001	**AH**: *add halfword*: packed 16-bit halfword unsigned addition of the contents of registers ***rs1*** and ***rs2*** (Comments: 8 separate 16-bit values in each 128-bit register)
xxxx0010	**AHS**: *add halfword saturated*: packed 16-bit halfword signed addition **with saturation** of the contents of registers ***rs1*** and ***rs2***. (Comments: 8 separate 16-bit values in each 128-bit register)
xxxx0011	**BCW**: *broadcast word*: broadcast the rightmost 32-bit word of register ***rs1*** to each of the four 32-bit words of register ***rd***
xxxx0100	**CGH**: *carry generate halfword*: for each of the eight 16-bit halfword slots, the register ***rs1*** is added to register ***rs2***, and the carry out bit placed in register ***rd***. (Comments: 8 separate 16-bit values in each 128-bit register)
xxxx0101	**CLZ**: *count leading zeros in words*: for each of the four 32-bit word slots in register ***rs1*** the number of zero bits to the left of the first non-zero bit is computed. If the word slot in register ***rs1*** is zero, the result is 32. The four results are placed into the corresponding 32-bit word slots in register ***rd***. (Comments: 4 separate 32-bit values in each 128-bit register)
xxxx0110	**MAX**: *max signed word*: for each of the four 32-bit word slots, place the maximum signed value between ***rs1*** and ***rs2*** in register ***rd***. (Comments: 4 separate 32-bit values in each 128-bit register)
xxxx0111	**MIN**: *min signed word*: for each of the four 32-bit word slots, place the minimum signed value between ***rs1*** and ***rs2*** in register ***rd***. (Comments: 4 separate 32-bit values in each 128-bit register)
xxxx1000	**MSGN**: *multiply sign*: for each of the four 32-bit word slots, the signed value in register ***rs1*** is multiplied by the **sign** of the word in vector ***rs2*** **with saturation**, and the result placed in register ***rd***. If the value in register ***rs2*** is zero, the corresponding 32-bit field in ***rd*** will also be zero. (Comments: 4 separate 32-bit values in each 128-bit register)
xxxx1001	**POPCNTH**: *count ones in halfwords*: the number of ones in each of the eight halfword slots in register ***rs1*** is computed. If the halfword slot in register ***rs1*** is zero, the result is also 0. Each of the results is placed into corresponding 16-bit slots in register ***rd***. (Comments: 8 separate 16-bit halfword values in each 128-bit register)
xxxx1010	**ROT**: *rotate bits right*: the contents of register ***rs1*** are rotated to the right according to the value of the 7 least significant bits (6 to 0) of ***rs2***. The result is placed in register ***rd***. Bits rotated out of the right end of the 128-bit contents of register ***rs1*** are rotated in at the left end.

Opcode [22:15]	Description of Instruction Opcode
xxxx1011	**ROTW**: *rotate bits in word*: the contents of each 32-bit field in register ***rs1*** are rotated to the right according to the value of the 5 least significant bits of the corresponding 32-bit field in register ***rs2***. The results are placed in register ***rd***. Bits rotated out of the right end of each word are rotated in on the left end of the same 32-bit word field. (Comments: 4 separate 32-bit word values in each 128-bit register)
xxxx1100	**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***. Bits shifted out for each halfword are dropped, and bits shifted in to each halfword should be zeros. (Comments: 8 separate 16-bit values in each 128-bit register)
xxxx1101	**SFH**: *subtract from halfword*: packed 16-bit halfword unsigned subtract of the contents of ***rs1*** from ***rs2*** (rd = rs2 - rs1). (Comments: 8 separate 16-bit values in each 128-bit register)
xxxx1110	**SFHS**: *subtract from halfword saturated*: packed 16-bit signed subtraction **with saturation** of the contents of ***rs1*** from ***rs2*** (rd = rs2 - rs1). (Comments: 8 separate 16-bit values in each 128-bit register)
xxxx1111	**XOR**: *bitwise logical exclusive-or* of the contents of registers ***rs1*** and ***rs2***

5 Expected Results

A **printed** project report including the goals, multimedia unit block diagram, design procedure, all testbenches, conclusions, and simulation results (both waveforms and results file) must be presented **at the start** of your 30-minute demonstration during a time slot assigned to your team by the TA.

The **electronic version of the report** must also be sent to the TA and Instructor **before the start of the presentation**.

Project presentation will not start without these printed and electronic documents submitted.

Project Demonstration Submission Period: November 29, demo last week of classes.

6 Milestone

The milestone for this project is due October 25. It should include complete code for the ALU, as well as a testbench. No knowledge of pipelining is necessary for this portion.