A simple MIPS-like processor

Yalun Wu

Department of Electronic Engineering, Denmark Technical University, Denmark
Electrical and Computer Engineering Department, University of Maryland, USA
yalunwu@gmail.com

ABSTRACT---In this paper, we describe the processor that we have designed during the semester. The instruction set for the processor is based on that of the MIPS instruction set which used 32 bits; the instruction set provides 31 different operations. The processor uses a 5 stage pipeline that is running on a clock of 8.33 MHz on the DE0 FPGA.

KEYWORDS: MIPS-LIKE, FPGA

I. INTRODUCTION

In this project, our objective is to build a processor that is pipelined and able to run on FPGA. The MIPS architecture provides a very simple approach to the processor. We, therefore, used the MIPS architecture as a basis for our development of the processor.

II. INSTRUCTION SET AND ASSEMBLER

We designed the instruction set to be in the style of MIPS instruction set [1]. We defined that there should be 32 General Purpose Register (GPR) for our design. The instruction for the processor is defined by a 32 bit word. The first 5 bits defines the operations that the processor will take. The following 27 bits will vary based on the operation, but they fall into 5 categories: (1) operation between 2 registers, (2) operation on a register and an input value of 22 bits, (3) single register operations, (4) comparison based operation which will take 3 registers e.g. branch if equal, and (5) interrupt and no operation which will take one value. The different type of operation is visualized below in figure 1.

Bit	2 Register	Reg Value	1 Register	Condition	NOP/Int
31					
30					
29					
28					
27	OpCode	OpCode	OpCode	OpCode	OpCode
26					0
25					0
24					0
23					0
22	Register	Register	Register	Register	0
21		i	0		0
20		i	0		0
19		i	0		0
18		i	0		0
17	Register	i	0	Register	0
16	0	i	0		0
15	0	i	0		0
14	0	i	0		0
13	0	i	0		0
12	0	i	0	Register	0
11	0	i	0	0	0
10	0	i	0	0	0
9	0	i	0	0	0
8	0	i	0	0	0
7	0	i	0	0	0
6	0	i	0	0	0
5	0	i	0	0	0
4	0	i	0	0	0
3	0	i	0	0	0
2	0	i	0	0	0
1	0	i	0	0	0
0	0	i	0	0	0

Figure 1 Operation and bit orders

All of the operation will be taken in terms of register to register operations. If the operation is between 2 registers the computed value will be stored on the register that is the second operand. For example, 00010|00010|00011|00000000000000000, which translate to ADD $R02, $R03, will add the value in register 2 and 3 together and store it in register 3. There is no overflow detection for the operations, it is assumed that the programmer will check the result and correct for such problems.

Conditional Branching operations take 3 registers as operands. The first two registers' values are used for the comparison operation. The value of the 3rd register is used as the branching location. For example, 11101|00011|11101|11100|000000000000 can be translated to "branch to register 28 if Register

3's value is greater than the value in register 29.

Additionally, we have chosen to have a load-store architecture. Memories are accessed indirectly, by first setting a register to the address of the memory, then loading the data from the memory.

The instruction set offers a wide range of operations and they are listed below.

Operation	OpCode	Type
NOP	00000	NOP
Move[1]	00001	2 Register
Add	00010	2 Register
Add Immediate	00011	Reg Value
Increment	00100	1 Register
Subtraction	00101	2 Register
Sub Immediate	00110	Reg Value
Decrement	00111	1 Register
Multiply[2]	01000	2 Register
Mut Immediate[2]	01001	Reg Value
Division	01010	2 Register
Div Immediate	01011	Reg Value
Shift left 1 bit	01100	1 Register
Shift right 1 bit	01101	1 Register
And	01110	2 Register
And Immediate	01111	Reg Value
Or	10000	2 Register
Or Immediate	10001	Reg Value
Nor	10010	2 Register
Not	10011	1 Register
Xor	10100	2 Register
Load	10101	2 Register
Store	10110	2 Register
Write[3]	10111	Reg Value
Swap	11000	2 Register
Branch	11001	1 Register
Branch if equal	11010	Conditional
Branch if not equal	11011	Conditional
Interrupt	11100	INT
Branch if greater	11101	Conditional
Branch if less	11110	Conditional

Figure 2 List of operations available

1. Copy data from one register to another
2. Only keeps the lower 32 bits
3. Write value to a register

III. System Architecture

We have designed the processor with a 5 stage pipeline. The pipeline uses the same structure as that of the classic RISC architecture. The 5 stages are fetch, decode, execute, memory access and write back [2].

Figure 3 RISC Pipeline

Fetch brings in the next instruction from the program cache. The Fetch stage also increments, or change it if branched, the program counter which will provide the next instruction. Decode stage reads the instruction and break it down. The instruction is broken down into chunk of OpCode, register numbers, and immediate values. The corresponding value grabbed from the GPRs, the storing registers, and the immediate values are feed into the Execute stage. The Execute stage will then computes the various values in the ALU. If the OpCode results in branching, the Execute stage will signal to the fetch state to update the program counter. If load and store are required, the corresponding flag will be raised. The computed data will also send to the Memory stage. The Memory stage accesses the RAM and will load/store values. If no memory access is needed, the values are passed to the Write Back stage which will store the values to the GPRs [2].

In order to handle data hazard, we have chosen to stall the pipeline. The data hazard is detected with in the Decode module. Once data hazard is detected, we will stall the pipeline until the operation on data is finished.

IV. Implementation

We implemented the processor in VHDL with the block diagram shown below. The different stages are separated into the different block of code as shown below. The instruction list stores the list of instructions that we will execute. In the fetch stage, for every clock cycle while "request fetch" is equal to '1', the program counter will increment and thus update next instruction. If "branching" equals to '1' then the address of the next instruction will be that in the "branching location."

In the decode stage, a huge case switch statement determines which operations to take based on the first 5 bits of the instruction. The operation will determine the data that goes into the execute stage and if it is ready for the next instruction or not. The data hazard

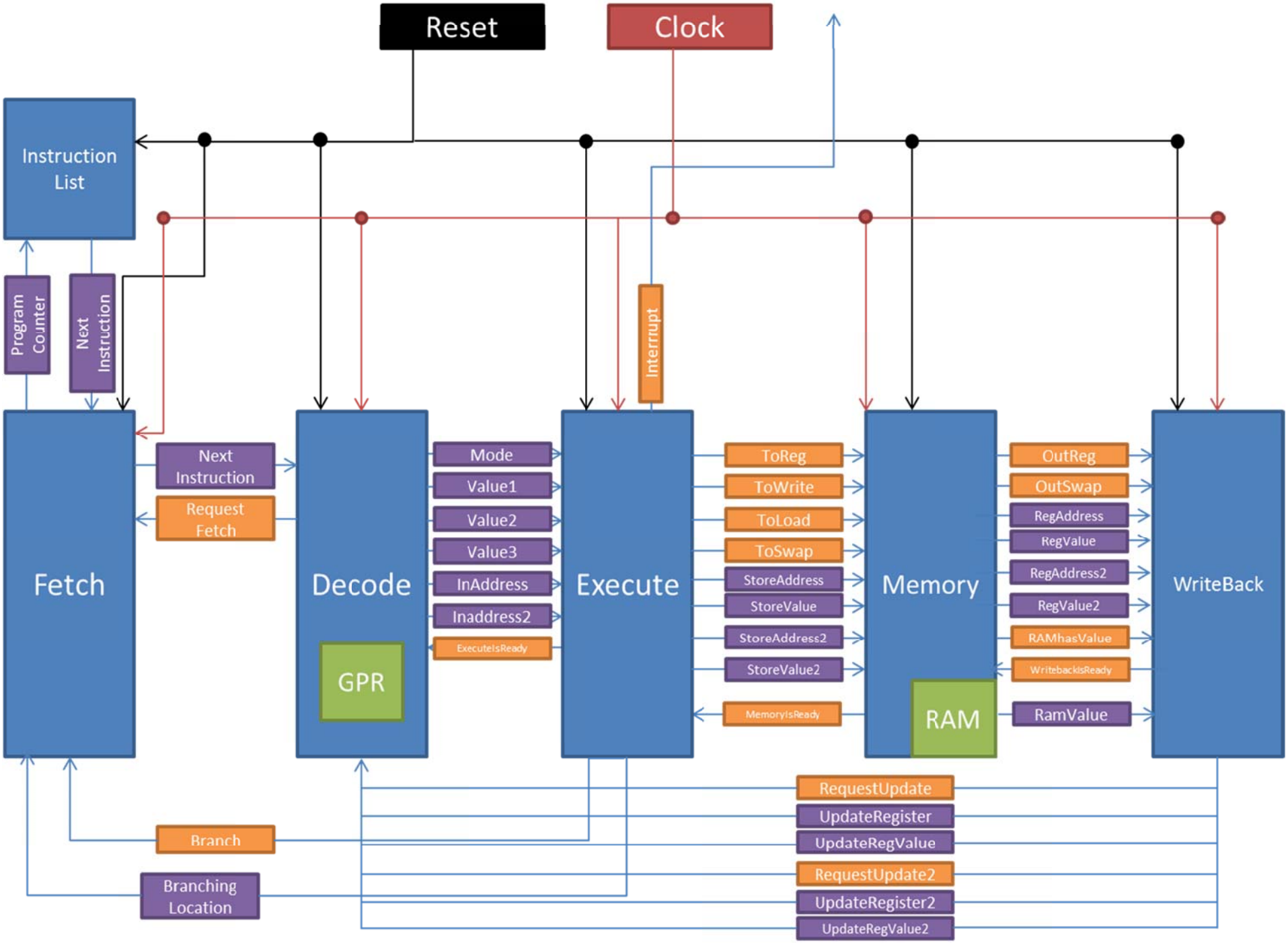
Reset
Clock
Instruction List
Program Counter
Next Instruction
Interrupt
Fetch
Next Instruction
Request Fetch
Decode
GPR
Mode
Value1
Value2
Value3
InAddress
Inaddress2
ExecuteIsReady
Execute
ToReg
ToWrite
ToLoad
ToSwap
StoreAddress
StoreValue
StoreAddress2
StoreValue2
MemoryIsReady
Memory
RAM
OutReg
OutSwap
RegAddress
RegValue
RegAddress2
RegValue2
RAMhasValue
WritebackIsReady
RamValue
WriteBack
RequestUpdate
UpdateRegister
UpdateRegValue
RequestUpdate2
UpdateRegister2
UpdateRegValue2
Branch
Branching Location

Figure 4 processor block overview (black-reset signal, red-clock signal, blue-code blocks, orange-signals, purple-data buses, green-internal parts)

Will be detected in here; it is detected by storing what registers are in use currently. The GPRs are in the form of an array of std_logic_vectors. The GPRs are updated when the "RequestUpdate" and/or "RequestUpdate2" equal to '1' with the corresponding value and address in "UpdateRegister" with "UpdateRegValue" and in "UpdateRegister2" with "UpdateRegValue2", respectively.

Execute Modules takes in the values from Decode and computes the results. If it is a branching operation and will execute a branch, the "Branching" signal will be "1" and "BranchingLocation" will be set. If it is any other operations the signal to update signal, the corresponding values and address will be passed along. We did not explicitly write a module for ALU since they can be simplified by using the mathematical symbols in the VHDL. In the case of division by 0, the resulting output will be 0. The same result will be shown if the subtraction results in a negative value. If "toReg" is '1' then a GPR value will be updated. If "toWrite" is '1' then a RAM value will be updated. If "toLoad" is '1' then a GPR value will be updated based on an address in RAM. If "toSwap is '1' then 2 values in the GPRs will be updated. If an interrupt operation is called the "interrupt" signal will be '1' for one cycle.

Memory modules takes in the value from execute and store it in the memory if it is requested. In the current design, we have a 1 kb RAM and it is implemented by using an SDPRAM module provided by the instructor. If a read from the ram is required, the resulting value, "RAMValue", is directly fed in the writeback module.

WriteBack module write the value to the GPR, which is located in the decode module.

We initially used the 50MHz clock to run the processor, but upon compilation we found some problems with running the processor in hardware. In Quartus, a program, TimeQuest, is provided which can be used to determine the slack and the worst case timing for the project. Upon examination of the timing result, we determine that we have to reduce the clock significantly because we have around -90ns of slack and this is a problem. In order to counter act this problem, we implemented a slower clock at a period 120ns, i.e.8.33 MHz, based on the 50MHz clock. Based on the data shown in Quartus, we used 8,325 out of the 18,752 total logical elements, 1663 registers, 8192 memory bits, and 6 Embedded Multiplier 9-bit elements

V. Example Programs And Result

We wrote 2 programs to test the correctness of the processor. The test programs are compiled via the assembler written for the project. First test result we did is a program that uses almost all the operations available in the instruction set and it was able to generate the correct result very easily. The purpose of this test is to check the functionality of the different operations and the correctness of the operations. The second program we write is to find the first 12 Fibonacci series number and it was able to generate those number correctly. The two programs are shown below.

Functional Check

```
NOP
NOP
ADDI $R00,55
WRT $R01,99
BNE $R16,$R13,$R28
WRT $R02,123
INC $R00
DEC $R02
WRT $R23,17
ADD $R00,$R03
MIN $R00,$R01
ADD $R02,$R04
ADD $R02,$R05
ADD $R02,$R06
MOV $R01,$R13
MINI $R06,2
BRH $R23
MUT $R04,$R05
MUTI $R13,500
INC $R07
WRT $R08,122
WRT $R09,2
DIV $R09,$R08
DIVI $R13,100
SLA $R09
SRA $R00
AND $R07,$R10
OR $R07,$R11
ANDI $R12,1
ORI $R15,1
XOR $R07,$R14
NOT $R10
NOR $R16,$R07
STE $R07,$R13
SWP $R10,$R13
ADDI $R14,100
BEQ $R14,$R15,$R07
```

```
LOD $R18,$R19
```

Fibonacci Series

```
ADDI $R00,1
ADDI $R01,1
ADDI $R28,27
ADDI $R29,100
ADDI $R30,10
STE  $R01,$R31
INC  $R31
MOV  $R01,$R02
ADD  $R01,$R02
STE  $R02,$R31
INC  $R31
MOV  $R01,$R03
ADD  $R02,$R03
STE  $R03,$R31
BGE  $R03,$R29,$R28
INC  $R31
MOV  $R02,$R01
ADD  $R03,$R01
STE  $R01,$R31
BGE  $R01,$R29,$R28
INC  $R31
MOV  $R03,$R02
ADD  $R01,$R02
STE  $R02,$R31
BGE  $R29,$R02,$R30
NOP
NOP
NOP
LOD  $R31,$R00
DEC  $R31
LOD  $R31,$R01
DEC  $R31
LOD  $R31,$R02
DEC  $R31
LOD  $R31,$R03
DEC  $R31
LOD  $R31,$R04
DEC  $R31
LOD  $R31,$R05
DEC  $R31
LOD  $R31,$R06
DEC  $R31
LOD  $R31,$R07
DEC  $R31
LOD  $R31,$R08
DEC  $R31
LOD  $R31,$R09
DEC  $R31
LOD  $R31,$R10
DEC  $R31
LOD  $R31,$R11
DEC  $R31
```

In order to output those results an additional module was written up. This module has a set of the same GPRs and was update the same time as the processor.

VI. Conclusion

We were able to successfully build a processor on a FPGA and it was able to run at 8.33 MHz. The processor runs on a 5 stage pipeline with 31 different operations. More work could be completed on the processor, including how to speed up the processor. Additionally, data hazard could be treated better by forwarding the data and or using Tomasulo algorithm.

The code for the program can be found in https://github.com/yalunwu/Processor.

VII. Bibliography

[1] J. F. Frenzel, "MIPS Instruction Reference," 10 September 1998. [Online]. Available: http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html. [Accessed 20 May 2014].

[2] D. A. Patterson and J. L. Hennessy, Computer Organization and Design, Morgan Kaufmann, 2011.