Computer Architecture BAMF Processor

GROUP 1-e

Ben Holtzman
AJ Granowski
Matt Moon
Tucker Osman

CONTENTS

EXECUTIVE SUMMARY

This report discusses the BAMF implementation of a 16-bit processor that can compute the least relatively prime integer to another integer using Euclid's algorithm. The report will analyze the design of the of the instruction set, the implementation, the Xilinx model, the testing procedures and methodology, and the final results of the processor. All relevant figures can be found in the appendices. The results of the processor will be discussed using the performance data collected in milestone 6. The report also discusses the shortcomings of the processor as well as the recommendations for improvements to the design.

INSTRUCTION SET DESIGN

The instruction set was designed to be convenient to implement as well as simple to write programs for. The instruction set was designed with the intention of using an accumulator design for the processor. A design in which there would be a single working register that would be used for doing operations on. Although there might be other supplementary registers used for storing other various data, the accumulator register is used in tandem with a stack to perform most operations. The instruction set was designed with 3 different types of instructions: F-type, I-type, and U-type. The F-type instruction is used when information needs to passed to the processor via the instruction. Some of the F-type instructions include: LUI, SLL, and SKIPIF.

Below is an example of an F-type instruction:

Instruction	Operand15	Type	Bit Pattern (little endian)
LUI	Upper 4 bits of 16-bit immediate	F-Type	0000|0001|0000[3:0 immediate]

Notice how the first 4 bits of the instruction are 0, this indicates the instruction is either an F-type or U-type. The next 4 bits of the instruction are the op-code for F-type instructions. The lower 8 bits are then used as a way to pass information that will change the behavior of the process based on the definition of the instruction.

The next instruction type created was the I-type instruction. This instruction type is used when large data values need to be passed to the processor via instructions. Some of the I-type instructions include: JUMPI, ADDI, and LOAD.

Below is an example of a I-type instruction:

JUMPI	12-bit immediate	I-Type	1000|[11:0 immediate]

Notice how at least one of the first 4 bits of the instructions is non-zero. This indicates that an instruction is an I-type. The upper four bits are then used as the op code for this instruction type. The lower 12 bits of the instruction can then be used by the programmer to pass data to the processor.

The last instruction type created was the U-type instruction. This instruction is used when all of the data used by the processor is implicit. This utilizes the accumulator design in a nice way. Instead of having to define registers or values to operate on, the U-type instruction uses only an op code to perform a process. This can allow up to 256 different U-type instructions (although the BAMF processor did not use nearly that many).

Below is an example of a U-type instruction:

ADDSTACK		U-Type	0000|0000|00010001

Notice how the first 4 bits of the instruction are 0, similar to the F-type instruction. For a U-Type instruction, the next 4 bits are also 0. This leaves the bottom 8 bits as the op code for the U-type instructions. This instruction in particular takes advantage of the accumulator register by using the

accumulator register as one of the argument registers and the destination register. The op code then defines that the value that will be added to working register comes from the stack.

The design approach taken by the BAMF design team proved to be a good one. From the original design document, only one instruction was not implemented. The versatility of the U-type instructions also allowed the team to later add useful instructions while the datapath was being designed. The U-type instructions only change control signals to execute, leaving few, if any, changes that needed to be made to the datapath.

IMPLEMENTATION

Once the instruction set had been mostly completed, RTL descriptions of each instruction could be written. The RTL descriptions of each instruction were then grouped by similarity and a unified RTL was created. It was from this unified RTL that the list of components was created.

COMPONENT DESIGN

Each component was defined with a list of inputs and outputs using the unified RTL. An English description of the operation of each component was also defined. A few components and their basic outlines are shown below:

- Arithmetic Logic Unit
    - Support for 13 different operations as specified by the unified RTL
    - Support for 2, 16-bit input and 1, 16-bit output
    - Can control various other components based on comparison results
- Control Unit
    - Can take an instruction and control all other components based on that instruction
- Register File
    - Can take input values and store them over a clock edge
    - Can turn on and off storage based on a control bit associated with each register
    - Can output the value stored in the internal registers immediately
    - Updates the internal values of the registers on the clock edge

It was using descriptions like this, that the components could later be created in Xilinx. Unifying the RTL made it simple to figure out what components would be needed and what they would need to support as inputs and outputs.

SUBSYSTEM DESIGN

There were 3 subsystems that were designed once the list of components was completed. The first was the memory subsystem. The memory is not only where the memory is stored but also functions as the stack which is used by many instructions. Complete functionality of this unit is crucial to the function of the overall datapath so a subsystem was designed so it could be appropriately tested. The second subsystem designed was the PC subsystem. This subsystem handles the program counter register. Another crucial register to the basic functioning of the datapath. Due to its importance, it was included as one of the subsystems. The last subsystem designed was the ALU subsystem. The ALU is used in almost every instruction, because of this, it needed to be verified as fully functioning

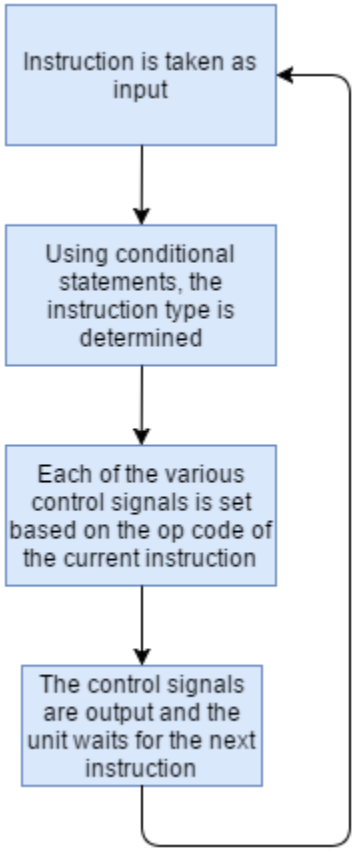
before it was integrated in the datapath. The ALU needed to work flawlessly so a subsystem was created around it to test functionality early in the Xilinx implementation.

DATAPATH DESIGN

Once the subsystems were verified to be working, the datapath was designed. The most important thing to the design team was the ease of implementation to support the use of so many instructions. Keeping this in mind we decided to use a single-cycle datapath. Then, using the unified RTL, the components could be connected so that each instruction could be performed. Support for each instruction was added one by one to the datapath. Eventually the datapath was completed and the control signals could be determined for each instruction. There are 46 separate instructions so various team members had to read over the control signal to verify they were correct.

XILINX MODEL

The Xilinx model was mostly completed in Verilog. Each component was designed in Verilog, with the subsystems and datapath designed using schematic capture. Using case statements, the design of some of the components can be trivialized. The control unit, all of the MUX's, and even the ALU function using case statements. Below is a flow chart outline of the Verilog code that was written for the control unit:



This design approach was taken for each unit, making the implementation of the basic components in Xilinx simple. The subsystems and datapath were designed using schematic capture. The subsystems were designed this way to make them easy to visualize, this would aid in the debugging process. The datapath was designed in schematic capture to match our datapath schematic from an earlier milestone and to also make it easier to visualize and debug during the testing process.

TESTING METHODOLOGY

COMPONENT TESTING

After any component was created, a corresponding test was made. These tests were pretty simple, they usually just ran through every combination of inputs and verified that the output is what was expected. Since the individual components were small, there weren't many inputs/outputs to check.

SUBSYSTEM TESTING

As previously described, we broke up the datapath into 3 subsystems. For each subsystem, we randomly chose a few possible instructions and set up any control signals/inputs that would be involved in that instruction. We checked all the outputs for each of those individual tests and made sure they were correct.

DATAPATH TESTING

The datapath test involved running every single instruction. That may sound simple, but it wasn't. We had to know that certain basic instructions like add/push/pop worked before we could reliably test any more complicated instructions. This meant these had to be tested at the top of the file. After we knew the basics worked, we put in tests that worked on general classes of instructions. For example, all the instructions involving memory were tested around the same time.

FINAL RESULTS

After our processor was working and tested, we collected the following performance results:

To store the whole program, the assembler required 110 bytes.
The total number of instructions that were executed was 102230.
The total number of cycles required to execute relPrime was 102230.
The average number of cycles is one because this is a single cycle datapath.
The cycle time for our design was 44.107ns (22.672MHz) but it can be sped way up.
The total execution time was 2,044,700ns. (2.0447ms)
The device utilization summary:

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Total Number Slice Registers	178	9,312	1%
Number used as Flip Flops	71		
Number used as Latches	107		
Number of 4 input LUTs	1,043	9,312	11%
Number of occupied Slices	601	4,656	12%
Number of Slices containing only related logic	601	601	100%
Number of Slices containing unrelated logic	0	601	0%
Total Number of 4 input LUTs	1,104	9,312	11%
Number used as logic	1,043		
Number used as a route-thru	61		
Number of bonded IOBs	98	232	42%
Number of RAMB16s	1	20	5%
Number of BUFGMUXs	1	24	4%
Average Fanout of Non-Clock Nets	3.73		

CONCLUSION

We managed to build a fully functioning processor in just a manner of weeks! Given more time, there are some changes we would make. Chief amongst them would be to make a more exhaustive test base, ours is admittedly minimal. If we broke up the integration of our processor into more steps, we could be more confident that it works as it should. Another change would be to make a more user-friendly I/O system. Ours only allows one input at a time and one output at a time. It wouldn't be terribly difficult to allow for more inputs and outputs, but it would make writing for our much more enjoyable.

Overall, I think it's safe to say that we made a BAMF processor.