CPE480 Assignment 3: Pipelined Tangled

Implementor's Notes

Cain Hubbard
Collin Lebanik
Nick Santini
Tristan Barnes
Department of Electrical and Computer Engineering
University of Kentucky
Lexington, Kentucky, United States

ABSTRACT

This is an pipelined implementation of a processor for the Tangled instruction set.

1 GENERAL APPROACH

Our AIK specification of Tangled is largely based on team 24's specification. See the Appendix A for the AIK specification.

We designed an ALU which would instantiate all the floating point operations. This would be done by making a call to the floating point instructions that we were given in the last assignment with some few changes. For this project we added NAN support for all of the floating point operations. Along with instantiating all of the floating operations our ALU also assigned the output based on the operation.

We broke our pipeline down into 4 stages inside one module called tangled. The names of the stages are as follows: Load, Decode, Execute, and Write-back. The first stage is the load stage which is used to determine where the instruction is coming from, either the pc or a branch instruction. Inside the load stage there are some functions that are used to determine if the instruction is a sys and a qat instruction which will be handled differently. The next stage in our pipeline is the decoding stage. This stage is used to determine what type of instruction is being used and it is figured out by using a bunch of functions to determine what instruction is being called. Value forwarding is also implemented inside the decoding stage. The next stage is the execute stage. This is where it is determined if a branch should be taken or not and where to go if the branch is taken. The writeback value is also determined in this stage. Lastly if a branch is taken then a bubble is implemented into the pipeline. The last stage in our pipeline is the writeback stage. This is where the writeback value is written into the regfile.

As recommended by Dr. Dietz, we made use of Verilog's `defines` to name ranges of bits we frequently use and the OP code for each instruction.

We decided to run our Verilog locally, so a few commands differ from Dr. Dietz's web interface. A Makefile is included to facilitate running the project. To run different assembly programs, the `readmemh` instructions must be changed to the appropriate file-names.

2 TESTING

Our testing consisted of carefully chosen test code that would cover most of the Verilog that we wrote. Our testbench includes a dumpfile macro with dumpvars that include the registers and wires we used in our Tangled module. Using the dumpfile along with those dumpvars allowed us to debug most of our program with the program GTKWave. With GTKWave, we could see the instructions and values in the order they were being executed, which when following along with the test assembly output from AIK, allowed us to follow the program and ensure our instructions were handled properly. Our Makefile, along with the testbench we created, allows us to use a macro that includes every test assembly file ending in `.tasm` that we created in order to produce dump files for all of our test cases at once. This allows us to view each of the different outputs for each of the given test cases in GTKWave individually. This means that the testbench does not need to be modified to run each of the different test assembly files.

Testing the Qat instructions is the most simple and straight forward of all of the instructions to test. This is because the Qat instructions are required to be trapped like a sys call. If the program halts after the specified Qat instruction then we know that it passed. Non-Qat instructions were added after the Qat instructions during testing to ensure that the program was actually halting and not just reaching the sys call at the end of our assembly instructions. Each of the different Qat instructions were given their own test assembly file. In order to run the specific Qat instruction, the GTKWave command along with the name of the specific Qat assembly dumpfile can be run. An example of the that would be:

```
$ gtkwave testing/qatOr.vcd
```

3 ISSUES

We ran into an issue with our value forwarding. If there was an instruction, say add after a branch, if the branch was taken then the add would not be executed (more specifically, the ALU add would be performed, but the value would not be written back to the register file). However, Since the add remains in the pipeline, any instruction that followed the branch's jump and used the register that the add would have written to would identify the add further down the pipeline and attempt to apply value-forwarding to use the result of that add instead of the current value for that register in the register file. This issue was resolved by applying an extra condition to our value-forwarding logic that ensured that the value being forwarded had not been "aborted".

The next issue was with our copy instruction. We ran into an issue where our values were not being copied right and we ran into

an instance where we had an infinite loop. The issue was that our copy instruction was writing back the wrong value. It was writing back the Rd value instead of the intended Rs value so we made a change to correct that.

A ENCODINGS

The instructions are divided into five different encoding formats, named Format 0 - 4, based on the number and bit-length of operands. Bit 15, named frmtA, specifies whether an instruction follows Format 0 or one of the other four formats by 1 and 0, respectively. Each of the remaining four formats are distinguished by the value specified by the bitfield 14 and 13, named frmtB. *Table 1* presents the frmtB encodings.

frmtB Encoding	Instruction Format
01	Format 1
10	Format 2
11	Format 3
00	Format 4

Table 1: frmtB Encodings

A.1 Format 0

Instructions encoded with Format 0 (i.e. frmtA = 1) contain a 4-bit operand, an 8-bit operand, and a 3-bit function code, named func0, that specifies the function of the instruction. The 4-bit operand is located in bits 12 - 9 and may specify one of the 16 main processor registers or a 4-bit immediate value, depending on the instruction. The 8-bit operand is located in bits 7 - 0 and may specify an 8-bit immediate value, an 8-bit PC-offset, or one of the 256 Qat coprocessor registers, depending on the instruction. The two most significant bits of func0 are located in bits 14 and 13 and the least significant bit is located in bit 8. The func0 bitfield is split up in order to allow the first 4-bit and first 8-bit operands in each format to be placed in the same location. This aids in simplifying microarchitecture design. *Figure 1* presents the Format 0 bitfields. *Table 2* presents the func0 encodings.
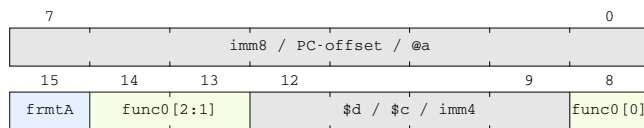

Figure 1: Format 0 Bitfields

A.2 Format 1

Instructions encoded with Format 1 (i.e. frmtA = 0, frmtB = 1) contain two 4-bit operand bitfields and a 5-bit function code, named func1, that specifies the function of the instruction. The first 4-bit operand is located in bits 12 - 9 and specifies a main processor register, which is the instruction's destination register if it contains two operands, otherwise it is simply the instruction's sole operand. The second 4-bit operand is located in bits 3 - 0 and specifies the instruction's second main processor register operand if the instruction contains two operands, otherwise the value is treated as a "don't

func0 Encoding	Instruction
0x0	lex
0x1	lhi
0x2	brf
0x3	brt
0x4	meas
0x5	next
0x6	had
0x7	*Undefined*

Table 2: func0 Encodings

care" and may hold an arbitrary value. (The assembler defaults to a second operand value of all zeros for such single-operand instructions). The func1 bitfield is located in bits 8 - 4. *Figure 2* presents the Format 1 bitfields. *Table 3* presents the func1 encodings. Notice that ALU and non-ALU Format 1 instructions can be easily distinguished by the MSB of the func1 field.

Figure 2: Format 1 Bitfields

func1	Instruction	func1	Instruction
0x00	not	0x10	jumpr
0x01	float	0x11	*Undefined*
0x02	int	.	.
0x03	neg	.	.
0x04	negf	.	.
0x05	recip	.	.
0x06	add	.	.
0x07	mul	0x17	*Undefined*
0x08	slt	0x18	load
0x09	and	0x19	store
0x0A	or	0x1A	copy
0x0B	shift	0x1B	*Undefined*
0x0C	xor	.	.
0x0D	addf	.	.
0x0E	mulf	.	.
0x0F	sltf	0x1F	*Undefined*

Table 3: func1 Encodings

A.3 Format 2

Instructions encoded with Format 2 (i.e. frmtA = 0, frmtB = 2) contain a single 8-bit operand and a 5-bit function code, named func2, that specifies the function of the instruction. The 8-bit operand,

which specifies a Qat register in all instructions, is located in bits 7 - 0. The func2 bitfield is located in bits 12 - 8. *Figure 3* presents the Format 2 bitfields. *Table 4* presents the func2 encodings.
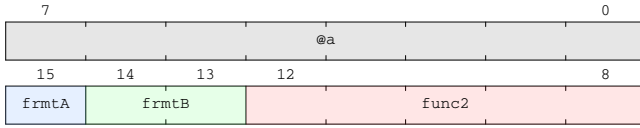

Figure 3: Format 2 Bitfields

func2 Encoding	Instruction
0x00	one
0x01	zero
0x02	not
0x03	*Undefined*
.	.
0x1F	*Undefined*

Table 4: func2 Encodings

A.4   Format 3

Instructions encoded with Format 3 (i.e. frmtA = 0, frmtB = 3) contain three 8-bit, Qat register, operand bitfields and a 5-bit function code, named func3, that specifies the function of the instruction. Format 3 instructions are the only 2-word (i.e. 32-bit) instructions in the Tangled ISA. The first operand is located in bits 7 - 0, the second in bits 23 - 16, and the third in bits 31 - 24. For instructions with only two operands, the third operand is treated as a "don't care" and may hold an arbitrary value. (The assembler defaults to a third operand value of all zeros for such two-operand instructions). The func3 bitfield is located in bits 12 - 8. *Figure 4* presents the Format 3 bitfields. Notice that the destination or single Qat register operand is aligned across Formats 0, 2, and 3. *Table 5* presents the func3 encodings. Notice that double- and triple-operand Format 3 instructions can be easily distinguished by the MSB of the func3 field.
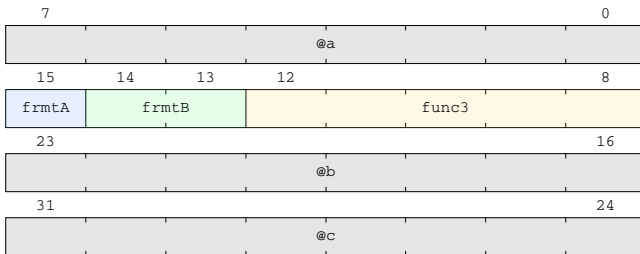

Figure 4: Format 3 Bitfields

func3	Instruction	func3	Instruction
0x00	ccnot	0x10	swap
0x01	cswap	0x11	cnot
0x02	and	0x12	*Undefined*
0x03	or	.	.
0x04	xor	.	.
0x05	*Undefined*	.	.
.	.	.	.
0x0F	*Undefined*	0x1F	*Undefined*

Table 5: func3 Encodings

A.5   Format 4

The only instruction encoded in Format 4 (i.e. frmtA = 0, frmtB = 0) is the sys instruction. Therefore, all of the remaining bits, bits 12 - 0, are treated as "don't cares" and may hold an arbitrary value. (The assembler defaults to a value of all zeros for these bits). Figure *Figure 5* presents the Format 4 bitfields.
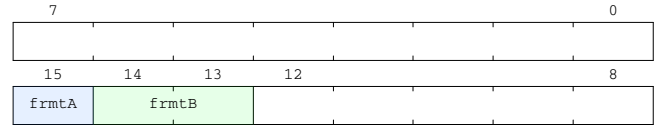

Figure 5: Format 4 Bitfields