CPE 480 Assignment 4: Pipelined Tangled + Qat

Implementor's Notes

Gerard Mijares, Christopher Butler,
Madankrishna Acharya, Malik Allahham
Department of Electrical and Computer Engineering
University of Kentucky
Lexington, Kentucky, United States
gami227@uky.edu, christopherbutler@uky.edu,
madan.acharya@uky.edu, malik.allahham@uky.edu

ABSTRACT

Assignment 4 requires students to design a pipelined processor which decodes and executes the Tangled ISA including the Qat instructions.

I. Tangled ISA

1 GENERAL APPROACH

We reused the encoding used in Assignment 3. The ISA encodings have been divided up into five different formats. Each format's bitfields are arranged such that similar operands align across all formats, thereby aiming to simply the decode logic that will be required by a microarchitecture that implements the Tangled ISA. Each format's encoding is defined in more detail in the following section.

2 ENCODINGS

The instructions are divided into five different encoding formats, named Format 0 - 4, based on the number and bit-length of operands. *frmtA,* specified by the value 0 or 1 in the 15th bit position of the instruction, separates the ISA into two main groups, *frmA0* and *frmtA1*. *frmtA1* is one group of instructions, and *frmtA0* is further split into four groups. Each of the four groups for *frmtA0* are distinguished by *frmtB*, the value 0-3 specified by bitfield 14 and 13 of the instruction. Table A depicts the *frmtA / B* encodings.

frmtA? Encoding	*frmtA@* Encoding	Instruction Format
1	*xx*	Format 0
0	01	Format 1
0	10	Format 2
0	11	Format 3
0	00	Format 4

Table A: Encodings

2.1 Format 0

Instructions encoded in 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 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 will aid in simplifying microarchitecture design. *Figure 1* presents the Format 0 bitfields. *Table 2* presents the *func0* encodings.

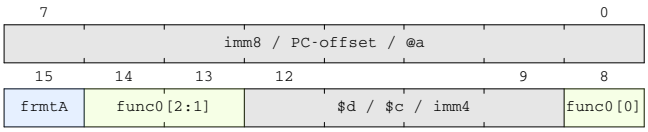

Figure 1: Format 0 Bitfields

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

Table 1: func0 Encodings

2.2 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 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 2* presents the *func1* encodings. Notice that single- and double-operand Format 1 instructions can be easily distinguished by the MSB of the *func1* field.

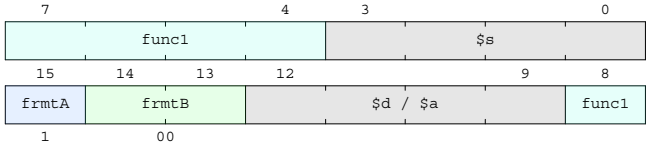

Figure 2: Format 1 Bitfields

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

Table 2: func1 Encodings

2.3 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 *func3*, 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.

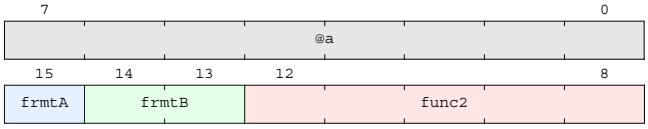

Figure 3: Format 2 Bitfields

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

Table 3: func2 Encodings

2.4 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.

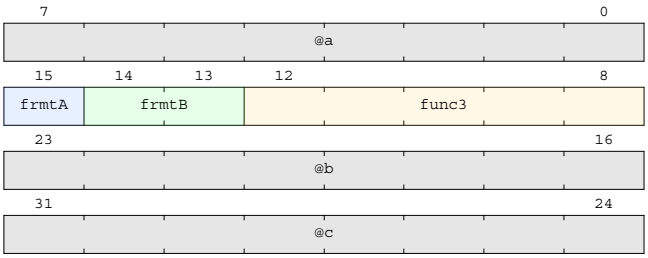

Figure 4: Format 3 Bitfields

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 4: func3 Encodings

2.5 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 2 presents the Format 4 bitfields.

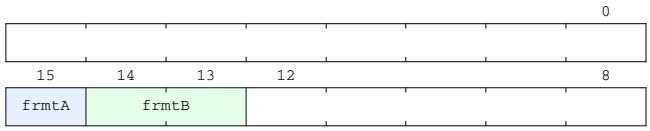

Table 5: func4 Encodings

3 PSEUDO INSTRUCTIONS

The AIK assembler specification fundamentally follows the pseudo instruction implementations suggested by the Assignment 1 handout. Additionally, the specification attempts to generate slightly denser machine code by using only the lex instruction to load 16-bit immediates when appropriate. That is, if the assembler decides that it must load a 16-bit address/immediate, it first checks to see if all of the high 16 bits are either all 0s or 1s, in which case it takes advantage of the fact that lex uses sign-extension and uses only lex to load the value, as opposed to the combination of lex and lhi.

II. *Pipelined Implementation*

1 GENERAL APPROACH

We reused our pipelined design from Assignment 3 which used an always block for each stage. The module *PTP* (Pipelined Tangled Processor) contains the four stages and ties them together to execute instructions in the *Tangled* set based on the AIK implementation described above. We modified stage 1 and 2 to only fetch one 16 bit instruction at a time since our design in Assignment 3 fetched both parts of the 32 bit Qat instructions during the same cycle in stage 1 which was not allowed for this assignment.

2 ALU Module

The module *ALU* receives three inputs: *op*, *x,* and *y.* A state machine decodes *op* into an operation performed on *x* and *y* and latches the result into an output *out.* This module performs all mathematical and logical operations in the Tangled instruction set.

2.1 Floating Point

The ALU includes instantiation of Dr. Dietz's floating point modules *fslt, fadd, fmul, frecip, i2f,* and *f2i*. Support has been added for not a number (NaN) by using the trinary operator to check for NaN conditions.

3 PTP Module

The processor module is divided into four stages: instruction fetch, instruction and register read, ALU and memory operations, and register write. Each stage is an *always* block with *if* statements to check what to execute for each clock cycle. Each stage writes to a buffer before the next stage. The next stage reads from this buffer to determine what to execute for each clock cycle. Only the fourth stage writes to the register file and only the second stage writes to memory, so there are not writing conflicts. The only exception to this is that *meas* and *next* write to the register file in stage 3, but dependencies are handled in stage 2, so there are no write conflicts.

We used hardware interlocks to prevent issues from read after write (RAW) dependencies. The second stage checks if the instruction uses a register that will be written to in an instruction in stage three or four. If the second stage detects a RAW dependency, it inserts a *noop* and tells stage 1 to resend the instruction. When the instruction arrives the second time, the dependency will have passed.

The *sys* instructions are passed to the next stage. When the *sys* instruction reaches the fourth stage, the processor halts.

The first 16 bit word for the 32 bit Qat instructions is stored in a register in stage 2. Then when the second part of the instruction comes in the next cycle, the instruction is passed onto stage 3. Stage 3 handles every aspect of the Qat instructions including reading from the register file, processing the instruction, and writing to both the Qat and Tangled register files.

As for implementing the Qat instructions themselves, for many instructions we were able to use algorithms described in the project assignment or in class. The only instruction that takes a small departure from this is next. Rather than zero-ing the first $d + 1 channels, we instead shift by that many channels and add $d + 1 back to the result we place in the Tangled register (if there is a next 1 at all).

III. *Testing*

1 ALU

The testbench for the ALU itself is not exhaustive. Instead, a single test case was conducted for each ALU operation and multiple test cases were made to test the NaN support. The ALU test resulted in 100% line coverage.

2 Processor

Our processor design was tested through running a sequence of instructions that included all of the Qat instructions. The instructions were organized to branch over a system call instruction if an operation was correct. If the operation was not correct, the branch would not be taken and the system call would halt the operation. We added an additional test to test data dependencies. The test ran successfully without halting until the end. We were not able to get the line coverage analysis to work for our processor testbench.

During testing, we found an issue with the pseudo *load* instruction. The *load* instruction works fine for loading values less than 128, but does not correctly load values greater than or equal to 128. We were unable to fix this issue, but we think the pseudo instruction has *lhi* load the wrong value in the top half of the register.

IV. *Submitted Files*

tangled.v - Verilog implementation of processor

tangled.aik - Tangled ISA

frecip_lookup.vmem - for floating point modules in VMEM0

ALU_TB.v - ALU testbench

test_cases.tasm - instruction sequence for testing

test_cases.vmem - for testing in VMEM1