Muffin Rockets

Group 2-C:
Ethan Petersen
Christopher Knight
David Lam
Kennan French

Table of Contents:

Executive Summary

The LoL processor and LoL assembly language are designed as a general-purpose 16-bit architecture. The processor is multicycle with memory-mapped I/O and 16 general purpose registers. The language supports 19 instructions, 4 of which are meant for I/O and interrupt handling. There are 5 different instruction types: A, C, M, J, and IH. A is meant for arithmetic, C is for arithmetic with large immediate values, M is for memory interactions and branches, J is for instructions which directly alter the program counter, and IH interacts with the interrupt architecture. Sixteen addresses in memory are reserved for I/O. For testing, we built unit tests for each component in our datapath that covered all of their respective functionalities. We also wrote integration tests that tested each portion of our RTL.

Introduction

The LoL architecture is designed as a 16-bit general-purpose processor and assembly language. In this report, we will explain the architecture beginning with a thorough description of the assembly language, covering instruction types, instructions, procedure calling conventions, and I/O. Then, we will describe the Xilinx model, beginning with control and ending with the data path. After that, we will go over the testing methods, both unit testing and integration testing. Finally, we will talk about the performance metrics and summarize the report. Appendices are added to go more in-depth to specifications, the design process, and other details.

LoL Assembly Language

Instruction Types

Our language has five instruction types: A, M, C, J, and IH. All instruction types begin with a 4-bit opcode. For A-type, the opcode is followed by a 4-bit Register Source, then a 4-bit Temporary Register, and finally a 4-bit Register Destination. These types are designed for performing arithmetic operations on values stored in registers. The Register Source and Temporary Register select the operands, while the 4-bit Register Destination defines the register to store the result.

For M-type, the rest of the instruction is filled with a 4-bit Register Source, then a 4-bit Register Destination, and a 4-bit Immediate. These instructions are designed for memory-interactions, such as loading from memory and storing to memory, and for branching.

For C-type, the instruction is filled with a 4-bit Register Destination and an 8-bit Immediate. These instructions are designed for arithmetic with a large immediate. The caveat here is that the operand register is also the destination register.

For J-type, the rest of the instruction is a 12-bit immediate. These are designed for jumps, i.e. instructions where the PC is written to.

IH-type has an opcode, followed by a 4-bit General Register Source, a 4-bit Selector, then a 4-bit Opcode. These instructions are meant for I/O and interrupt handling. There cannot be two consecutive IH-type instructions that are not “Clear”. If two need to be executed consecutively, an “add $0 $0 $0” instruction should be placed between them.

Instructions

We will present the instructions in ascending order of the opcodes (i.e. from 0000 to 1111). First, “add” is an A-Type that will add the contents of the source register and temporary register then store the result in the destination register. Then “sub” will subtract the temporary register value from the source register and store into the destination register. The “and” instruction will perform a logical and on the source register and temporary register values then store into the destination register. The “or” instruction will perform a logical or on the source register and temporary register values then store into the destination register.”Addi” will add the immediate to the destination register then store into the destination register. “Mv” will copy the value in the interrupt register and store it into the destination register. The general register destination is determined by the 4-bit destination register in the instruction. “Clr” will clear a selected bit in the interrupt register, based on the 4-bit selector in the instruction. “Read” will load input data into a selected location in memory. This location is based on the 4-bit selector in the instruction. “Display” will load data from a specified memory location into the display register. This location is based on the 4-bit selector in the instruction also. “Ori” will perform a logical or on the immediate with the selected general register. The immediate is zero-extended. “Slt” will store a 1 in the destination register if the source register is less than the temporary register; otherwise, it will store a 0 in the destination register. “Lw” will load the value stored in memory at source register + sign-extended immediate, into the destination register. “Sw” will store the value in the destination register into the source register + sign-extended immediate. For both instructions, the source register is selected by the 4-bits following the opcode, then the destination register is determined by the next 4-bits. “Lui” will zero-extend an 8-bit immediate value and store into the destination register. “Beq” will overwrite the program counter with program counter + sign-extended immediate if the two source registers are equal. “Bne” will overwrite the program counter with program counter + sign-extended immediate if the two source registers are not equal. “Jal” will overwrite the program counter with the top four bits of the program counter followed by the 12-bit immediate and store the original program counter into the return address. “J” will only overwrite the program counter with the top four bits of the program counter followed by the 12-bit immediate. “Jr” will overwrite the program counter with the value stored in the source register.

Assembly Conventions

Callers are required to:

a. Create a stack to save their current return address.
b. Fill registers 2 and 3 with arguments meant for the procedure being called.
c. Fill registers 10-12 with values meant to persist through the procedure call.
d. Update the return address register with the address of the instruction after the jump to the procedure call.
e. Jump to the procedure call.
f. Load values from the stack into the corresponding register.
    i. Specifically, the return address should be put into register 11.
g. Undo the stack.
h. If relevant, use register 12 as the return value of the procedure call.

Callees are required to:

i. Save values in the save registers to memory.
    i. Note, callees are free to use temporary registers.
j. Load the save values from memory to registers before returning.
    i. These will not be modified throughout the rest of the procedure.
k. Use register 12 as storage for a single return value.
l. Jump to the return address.

For Interrupt handling, two non-clear warp instructions should be separated with a single no-op (add $0 $0 $0).

If branch is out of range, an inverse conditional branch should branch over a jump instruction. For example, if a branch intended to go forward 8 instructions, the following should be used instead:

```
bne    ___    ___    # Branch out of range for beq.
j      ___    ___    # Address to jump to.
# Line bne should jump to
```

Input/Output

For our architecture, I/O is interrupt-driven so when a bit is flipped in the interrupt hardware, this bit is stored in the interrupt register, and when control executes an instruction fetch, it will store the program counter into an exception program counter, and then the program counter is overwritten with the interrupt handling location in memory. We initially load memory with interrupt handling code starting at address 0x3000. This code will move the interrupt register to a temporary register, and then it will jump to other interrupt handling code depending on the interrupt. Input will use a “read” instruction, which will load the input data into a place in memory that corresponds to the interrupt hardware. Locations 0x2ff0 through 0x2fff are reserved for input/output. Once the interrupt is handled, a “clr” instruction will clear the corresponding bit in the interrupt register and reset the program counter to the value stored in the exception program counter. If output is to be displayed, then the “display” command should be used in which case it will load the selected I/O address from memory into the display register. If another interrupt occurs while an interrupt is being handled, this will be recorded in the interrupt handler and once the current interrupt is handled, the next interrupt will begin to be handled.

Xilinx Model

Control

Our State Control Unit was comprised of two states. We had a current_state, which told us the current stage of the current instruction. We also had a next_state, which told us what the next stage of the instruction was going to be after the current stage finished.

Our first two cycles were the same for every single instruction. We had an Instruction Fetch stage which would load the instruction out of memory. Our second cycle was an Instruction Decode. This stage allowed us to check the Op-Code of the Instruction and interpret what to do with it.

Our third cycle depended on the Op-Code. We would interpret what type of instruction we were doing and then go to a specific control state.

Our fourth cycle was fully dependent on the third cycle and in one case the Op-Code. Some of our instructions, like our branches and our regular jump only needed three cycles. Our arithmetic instructions usually needed four. Also, our load word and store word third cycle would check the Op-Code again in the third cycle to set the fourth cycle to the correct instruction.

We only three five cycle instructions. Warp, Load word, and Store word all needed five cycles in our architecture.

At the end of all the cycles form a particular instruction, we set the next_state to be an Instruction Fetch. This allowed us to continue going through each instruction.

For a much more in-depth look at control and our diagram, take a look at our State Control Unit V2 and our Register Transfer Language Diagram.

Data Path

In this section, we'll describe the Xilinx data path. First, we will describe the components that make up the datapath. There is a program counter register (PC) that takes a 16-bit input and outputs a 16-bit value, with three control signals: PCWriteBeq, PCWriteBne, and PCWrite. This feeds into a mux that chooses between PC, an ALUOut register, and a decoder value, whose output then goes in as a read address in memory. A 3-input mux will choose between ALUOut, data from input, and a register value, choosing one to feed into memory. We use a block memory component to store both instructions and data. This takes a 14-bit read address and a 16-bit write value, and a 1-bit control signal called MemWrite. There is a display out register that takes a 16-bit input and outputs a 16-bit value. This is controlled by a 1-bit lcdWrite signal. There is an instruction register which reads a 16-bit value, termed Instruction, from block memory and outputs Instruction[15:12], Instruction[11:8], Instruction[7:4], and Instruction[7:0]. This is controlled by a 1-bit IRegWrite signal. There are two sign extending components: one that sign extends a 4-bit value to 16-bits and one that extends an 8-bit value to 16-bits. Also, there is a component that zero extends an 8-bit value to 16-bits. There is a general register file that holds 16 registers. This takes three 4-bit inputs and one 16-bit input, with two 16-bit outputs, and one 1-bit control signal: GRegWrite. Two of the 4-bit inputs are for read addresses and the third is for the write address. The 16-bit input is the data to be written to the write addressed register upon a flipped control signal. A 4-input mux will choose the address to write to, and a 4-input mux will choose the data to write in. The two outputs of the register file go into two separate registers, A and B. The output of register A feeds into a 2-input mux, ALU Source A, that also has the PC output as the other input. This mux outputs to the ALU. The output of register B feeds into a 4-input Mux, ALU Source B, that has two constants: 0 and 1, along with another value that's either a sign-extended immediate or zero-extended immediate. The output of the ALU feeds into both a 6-input mux and an ALUOut register. The mux has the value from register A, the value from the ALU, the output of ALUOut, a constant that represents the interrupt handling address, and the top four bits of PC followed by the bottom 12-bits of Instruction as inputs. The output goes directly to PC, chosen by a 2-bit control signal called PCData. Another mux will choose between both of the sign extended values to be passed along to the ALU Source B mux.

Testing

Each individual component in the processor is rigorously unit tested. In the Appendices, we go in-depth as to the unit tests for every component. Additionally, we performed integration tests for the model. These tests follow the RTL design: testing the first two phases, and then testing the phases of each instruction.

Performance

Our processor is blazingly fast. The processor's average CPI is 3.335 and it only takes 223,000 Bytes to store relPrimes and euclid's. Only 61,292 instructions are executed in relPrimes for a total of 204,404 cycles, taking 5.519 ms. Below is the timing summary for the Xilinx Model:

Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Total Number Slice Registers	606	9,312	6%	
Number used as Flip Flops	574			
Number used as Latches	32			
Number of 4 input LUTs	1,336	9,312	14%	
Number of occupied Slices	906	4,656	19%	
Number of Slices containing only related logic	906	906	100%	
Number of Slices containing unrelated logic	0	906	0%	
Total Number of 4 input LUTs	1,441	9,312	15%	
Number used as logic	1,336			
Number used as a route-thru	105			
Number of bonded IOBs	27	232	11%	
Number of RAMB16s	15	20	75%	
Number of BUFGMUXs	2	24	8%	
Number of RPM macros	2			
Average Fanout of Non-Clock Nets	3.17			

Conclusion

If you're a developer working in a 16-bit environment, please consider the LoL processor and LoL assembly language. We offer a general-purpose solution with support for pseudo-instructions, vastly multiplying the possibilities for an efficient workflow. Coupling that with our speedy multi-cycle processor will surely put your company ahead of the competition. With 16 general-purpose registers, 19 core instructions, and 16 I/O addresses, writing programs has never been so easy.

Appendices

Appendix A: Register Details

Register number	Register name	Purpose
0	$0	Constant zero
1	$a0	Function argument
2	$a1	Function argument
3	$t0	Temporary variable
4	$t1	Temporary variable
5	$t2	Temporary variable
6	$t3	Temporary variable
7	$s0	Saved variable
8	$s1	Saved variable
9	$s2	Saved variable
10	$sp	Stack pointer
11	$ra	Return address
12	$rv	Return value
13	$at	Assembler temporary
14	$k0	Interrupt Special Register
15	$k1	Interrupt Special Register

Appendix B: Instruction Details

Op code	Instruction	Instruction name	Type	Instruction format
0000	add	Add	A	$rd, $rs, $rt | $rd = $rs + $rt
0001	sub	Subtract	A	$rd, $rs, $rt | $rd = $rs - $rt
0010	and	And	A	$rd, $rs, $rt | $rd = $rs & $rt
0011	or	Or	A	$rd, $rs, $rt | $rd = $rs | $rt
0100	addi	Add immediate	C	$rd, IMM | $rd = $rd + IMM
0101	warp	Warp	IH	Varies (see below)
0110	ori	Or immediate	C	$rd, IMM | $rd = $rd | IMM
0111	slt	Set less than	A	$rd, $rs, $rt | $rd = 1 if ($rs < $rt), 0 otherwise
1000	lw	Load word	M	$rd, $rs, IMM | $rd = Mem[$rs + IMM]
1001	sw	Store word	M	$rd, $rs, IMM \| Mem[$rs + IMM] = $rd
1010	lui	Load upper immediate	C	$rd, IMM | $rd = IMM(upper 8 bits)
1011	beq	Branch equal	M	$rs, $rt, IMM | PC = PC + IMM if $rs = $rt.
1100	bne	Branch not equal	M	$rs, $rt, IMM | PC = PC + IMM if $rs != $rt.
1101	jal	Jump and link	J	IMM| $ra = PC + 1 | PC = PC[15:12] + IMM
1110	j	Jump	J	IMM | PC = PC[15:12] + IMM
1111	jr	Jump register	J	$rs | PC = $rs

Warp function	Function code	Instruction	Instruction format
Move	0000	mv	$rs xxxx 0000
Clear	0001	clr	xxxx selector 0001
Read	0010	rd	xxxx xxxx 0010
Display	0011	dsp	xxxx location 0011