ECE151A : Team Aardvark - "BAG" 8-bit Processor Design

Andy Jeong, Gordon Macshane, Brenda So

Contents

1 ABSTRACT

Employing a subset of MIPS ISA, an 8-bit processor – BAG – was designed using Verilog language. BAG implements a RISC architecture, with 13 core instructions and 1 pseudo instruction. BAG offers a preassembler and assembler that works on the C language. Each instruction is set to 1-byte, or 8-bit, length, and the main memory is byte addressable. There are 4 addressable registers: $s1 (00), $s2 (01), $sp (10) and $ra (11). Two other registers in the register file, $slt_0 and $slt_1, cannot be addressed directly, but may be well utilized for jumping (beq instruction). All registers are initialized at 0, while the $sp pointer is initialized at 0xFF (255 in decimal), which is the top of the memory stack. The memory has the ability to store 76 instructions, 4 global variables and the rest are allocated for the stack. A recursive and non-recursive code is used to demonstrate the functionality of the code

2 MEMORY

The memory is byte addressable, since all registers can only carry 8 bits and each instruction has 8 bits. There are three sections of the memory stack, the text (0x00 - 0x4B), the data (0x4C - 0x4F), and the stack (0x50 - 0xFF). The allocation of text section is determined to be 76 bytes of instruction because the programs presented in general involve at most 28 lines, hence 76 lines would be suitable for a program that is meant for our ISA. The data section is limited to only 4 global bytes in general no more than 4 arguments will be taken for most programs (unless very large programs, which are not considered for this processor).

Since the fact that a pseudo-instruction–load address("la")–needs to handle loading of data into stack, the assembler was written for this particular processor. If a programmer declared a global variable in the data section of the memory, the pseudo instruction would need to be expanded into the instructions available in the core instruction set. In the worst case scenario, each "la" command would need to be replaced by 16 core instructions. Since the text section can only occupy at most 76 bytes of instructions, there could only be at most 4 bytes for global data, assuming each data is one byte.

3 ASSEMBLER

In order to deliver the assembly code to Verilog to compute, the assembly language had to be converted to binary machine code. To that end, a set of assembler with a linker was implemented in C language. Two assemblers were used to demonstrate two major processes of converting assembly to machine code.

The first program, preassembler.c, takes in the ISA and converts the instructions to that in our ISA. First, all global variables are converted into binary code and stored in the first section of the linker file. As for .text section, first all psuedo instructions are scanned and converted into core instructions. The existence of psuedo instructions allows the programmer to declare global variables more easily. Second, the file is re-scanned and all the labels are given an absolute address. Third, the file is scanned again for detecting labels at J instructions, which is then converted to relative addresses. This is to ensure that the output linker file would be in a format such that assembler.c can manipulate the core instructions smoothly and output a binary file.

The second program, assembler.c, the linker file is converted into machine code. Each instruction is cross referenced and each line outputs an 8 byte binary code according to our Green Sheet (See

last page). The output is a 256 line 8-bit binary file, with lines 1 - 76 as the area for instructions, and 77 - 80 as the area for global variables, and the rest is the stack.

4 ISA OVERVIEW

4.1 Core Instruction

BAG is a RISC architecture with 13 core instructions and 1 pseudo instruction. There are four types of commands – R, I, J and JR, each having similar instruction formats. These similar formats allow an easier implementation of the processor. The functionality of R and I instructions are expanded with the addition of funct code. Only 3 bits are used to indicate beq and jal such that instructions can jump from -16 to 15 bits away. By the aforementioned restrictions, the JR instruction is restricted to one specific byte pattern, but this compromised is justified by the range of jumps that can occur and the range of R and I instructions offered.

4.2 Pseudo Instruction

The pseudo instruction – load address – allows users to load in global variables more easily. When a psuedo instruction is detected by the preassembler, the preassembler matches the label of load address to the respective address in the memory stack. The preassembler reads the address from right to left and depending on whether it is a one or a zero, decides whether to only shift the value in the register left or both shift left and add an immediate by 1. The process continues until the whole address is read, For instance, for the first global addres, which is number 77, the following code is generated from the pseudo code:

```
sl $sp $sp
sl $sp $sp
addi 1 $sp
sl $sp $sp
sl $sp $sp
sl $sp $sp
addi 1 $sp
sl $sp $sp
addi 1 $sp
sl $sp $sp
sl $sp $sp
addi 1 $sp
```

5 TOOLCHAIN

The compilation and execution of this program is handled by a Makefile, which contains commands to compile and run the assembler in C and Verilog processing codes. The protocal “asm in the makefile is for assembling using preassembler.c and assembler.c. At this stage, preassembler.c takes in an input from the user, and outputs a “link file, which is the input for assembler.c, where the link file produced contains translation of pseudocodes, “jal and .data variables. This link file is then converted to memory addressable binary file through assembler.c. After processing the assembler, a file called 'output.bin' is created for Verilog to process. With the protocol “run , one can run Verilog main_tb.v testbench, where memory module takes in the binary file for instructions. A waveform could also be simulated by the protocol “wave and this is realized through GTKwave simulator program. One needs Icarus Verilog and GTKwave programs to fully see the functionalities of this processor.

6 PROCESSOR DIAGRAM

The stages of procession are similar to MIPS:

1. FETCH
2. DECODE
3. EXECUTE
4. MEMORY ACCESS
5. WRITE-BACK

Features of processor are visually shown below, with inputs and outputs are clearly indicated in blue before and after each module.

1. Instruction memory and data memory in a single unified memory block (Von Neumann Architecture)
2. Instruction Decoder
3. Overall Control and ALU Control Unit
4. ALU that performs nand, add, compare, equality check, shift left and shift right)

pcAddrOut
00000001
ALU R
pcAddrOutPlusOne
jctrl
JRctrl
pcAddrOut
rsData
pcAddrOutPlusimm
pcAddrNext
nextctrl
pcAddrOutPlusOne
pcAddrOutPlusimm
jalAddr
Jump Decision
rsData (returnAddr if jrctrl = 1)
pcFinalOut
pcFinalOut
PC
pcAddrOut
11111111
jimm8
jumpCtrlToMux
ALU 0
jumpCheck8
Sign Ext
Sign Ext
0
1
mux.JumpFlag
Control Unit
memWrite
memRead
memToReg
ALUop
ALUsrc
regWrite
JRctrl
shtCtrl
funct
ALU Control
ALUctrlbits
jumpFlag_ALUctrl
beqSatisfied
jctrl
jumpFlag_ALUctrl
beqctrl
jimm5
jumpCheck2
funct
Inst Reg
inst2CtrlUnit
rtAddr
rsAddr
beqctrl
rt (read reg 1)
rs (read reg 2)
rt (read data 1)
rtData
s1_data, s2_data, sp_data, rs_data
Register File
rs (write reg 2)
rs (read data 2)
rsData
dataToWrite
rtAddr
returnAddr
rsWriteAddr
ractrl
instruction
Instruction Memory
jimm2
Sign Ext
jimm8
muxALUsrc
ALU R
shtReg
ALUresult
Address
dataMemWrite
Data Memory
readData
pcAddrOutPlusOne
ALUresult
instruction (current pc)
pcAddrOut
jimm5
jumpCheck2
funct
Team Aardvark: 8-bit Computer
Members: Andy Jeong, Gordon Macshane, Brenda So
ECE151A Spring 2016
Professor Marano, Cooper Union
Last updated: 5/7/2016

7 DESIGN CONSIDERATIONS

1. **RISC design**: Each type of instruction has a specific format to adhere to. In particular, the R-type instruction and I-type instruction are very similar in terms of structure. This adheres to the principles of simplicity favoring regularity and making the common case fast, because the formatted instructions would be easier on the programmer and the similarity between instructions makes the processor simpler and easier to design.

2. **Jumping instructions**: Good design demands good compromise. The jumping instructions, beq and jal, have only 3 opcode while the R and I type instructions have 3 opcodes and 1 funct code. Although the difference in the length of opcode would make the design of the input of control unit slightly more complicated, it actually allows the user to jump by a -16 to 15 byte offset from the current program counter. If 4 bits are used instead, the user could only jump an eight byte offset.

3. **beq command**: The beq command works different than that of MIPS architecture. Again, this is a compromise to be made in order for jumping to be able to occur across 32 bytes of instructions. The beq instruction compares the two unaddressable registers – slt_0 and slt_1, and checks for equality between the two registers. The two registers can be manipulated through slt_0 and slt_1 command.

4. **Jumping combinational logic**: In essence, there are 3 kinds of jumps our ISA handles: unconditional jumps (jal), conditional jumps (beq) and jump register(jr). There are also 3 options for the next address in program counter, namely PC+1, PC+immediate and value stored in $ra. Hence we build a combinational logic with MUXes to choose between the three options under 3 different conditions.

5. **Pseudo instruction**: In the case where the programmer does not know how the memory is organized, they could use pseudo instruction to access to global variables. The psuedo instruction translation is handled by the preassembler. We use shift left and addi to compose an efficient algorithm to load the address into a register.

8 EXTRA CREDIT

We attempted to implement an 8 byte cache to data memory, although we could not successfully do so. We wanted to take a step further in our design by implementing a direct map caching with a write back policy, since direct map cache is less expensive than full associative cache in terms of comparisons necessary to build the cache, and a writeback policy such that the CPU do not need to communicate directly to the ROM. The failure to meet this goal could be attributed to the timing issues and race conditions that were encountered while the caching mechanism.

9 SAMPLE CODE

In this section, a non-recursive code (multiplication) and a recursive code (summation) is written to demonstrate the functionality of BAG ISA.

9.1 NONRECURSIVE CODE

9.1.1 Non-recursive Code in C

```
int mult(int a, int b)
  {
      int c = a * b;
      return c;
  }
```

9.1.2 Non-recursive Code in Assembly

```
.data
.byte num1 2
.byte num2 3

.text
la $sp num1                    //load address of num1 into stack pointer
lw 0 $s1                       //load num1 into s1
lw 1 $s2                       //load num2 into s2
nand $ra $sp                   //move stack pointer up
sw 0 $s1                       //save num1 into data memory
lw -1 $s1                      //load zero into s1
addi -1 $sp                    //move stack pointer down
jal MULT                       //go to multiplication
ADDITION: lw 1 $s2             //load num1 into s2
          add $s2 $s1          //$s1 = $s1 + $s2
          lw 0 $s2             //load num2 back into s2
MULT: lw -1 $ra                //load zero into $ra
      addi -1 $s2              //$s2 = $s2 - 1
      sw 0 $s2                 //save $s2 into memory
      slt_0 $ra $s2            //check whether $s2 < 0
      beq ADDITION             //if equal, branch to addition
      addi -1 $sp              //move stack pointer down
sw 0 $s1
```

9.2 RECURSIVE CODE

9.2.1 Recursive Code in C

```
int sum(int n)
  {
      if (n > 1){
          return n + sum(n-1);
      }
      return 1;
  }
```

9.2.2 Recursive Code in Assembly

```
.data
.byte num 5

.text
la $sp num              //load num into memory
lw 0 $s1
nand $s2 $sp            //push stack pointer back up
addi -2 $sp             //move stack pointer down by 2 bytes
jal SUM
jal DONE
SUM: addi -2 $sp        //move stack pointer down
     sw 0 $ra           //save return address
     sw 1 $s1           //save accumulated value
     slt_0 $s2 $s1      //Check whether s2 < 0
     beq LOOP           //If so go to LOOP
     addi 1 $s1         //If not, add 1 to $s1
     addi 1 $sp         //move $sp up by 2 bytes
     addi 1 $sp
     jr $ra             //jump to address stored in $ra
DONE: jal DONNER        //jump to DONNER
LOOP: addi -1 $s1       //$s1 = $s1 - 1
      jal SUM           //jump to SUM
      lw 0 $ra          //load $ra
      lw 1 $s2          //load $s1
      addi 1 $sp        //move $sp up by 2 bytes
      addi 1 $sp
      add $s2 $s1       //$s1 = $s2 + $s1
      jr $ra
DONNER: sw 0 $s2
```

10 APPENDIX I. REFERENCE SHEET

BAG Reference Data

CORE INSTRUCTION SET

NAME,MNEUMONIC		FORMAT	OPERATIONS	OPCODE	FUNCT
ADD	add	R	R[rt] = R[rs] + R[rt]	000	1
NAND	nand	R	R[rt] = NAND(R[rs],R[rt])	001	1
SET LESS THAN (slt_0)	slt_0	R	R[rs] < R[rt] ? 1 : 0	010	0
SET LESS THAN (slt_1)	slt_1	R	R[rs] < R[rt] ? 1 : 0	010	1
SHIFT LEFT	sl	R	R[rs] = R[rt] << 1	011	0
SHIFT RIGHT	sr	R	R[rs] = R[rt] >> 1	011	1
LOAD WORD	lw	I	R[rs] = Mem(R[sp] + immediate)	100	0
SAVE WORD	sw	I	Mem(R[sp] + immediate) = R[rs]	100	1
ADDI	addi	I	R[rs] = R[rs] + immediate	101	0
BRANCH IF EQUAL	beq	J	if R[slt_0]=R[slt_1]; PC = PC+immediate	110	n/a
JUMP AND LINK	jal	J	R[ra]=PC+1; PC = PC+immediate	111	n/a
JUMP REGISTER*	jr	JR	PC = R[ra]	101	1
LOAD ADDRESS**	la		R[rs] = label		

* jr only jumps to R[ra] and has opcode 10110011

** Pseudo Instruction loads address using shift left and add immediate

BASIC INSTRUCTION FORMAT

Type	7	6	5	4	3	2	1	0
R	opcode			func	rt		rs	
I	opcode			func	immediate		rs	
J	opcode			PC relative immediate				
JR	1	0	1	1	0	0	1	1

REGISTER NAME, NUMBER, USE, CALL CONVENTION

REGISTER NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$s1	00	Saved Temporary	Yes
$s2	01	Saved Temporary	Yes
$sp	10	Stack Pointer	Yes
$ra	11	Return Address	Yes