Aardvark - 8-bit Computer Processor Design

JEONG Andy, MACSHANE Gordon, SO Brenda

Contents

1 ABSTRACT

Employing a subset of MIPS ISA, an 8-bit processor was designed using Verilog language. Each instruction is set to 1-byte, or 8-bit, length, and the main memory is word addressable while the instruction 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.

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 64 bytes of instruction because the programs presented in general involve at most 28 lines, and as the maximum 64 was enough. 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, an assembler with a linker were implemented in C language. The first portion of the assembler takes in the assembly file "input.s" and outputs a link file, which is then processed through our second part, which then fully converts it to absolute memory-addressable binary. The output of this is then fed into the Verilog code.

4 PROCESSOR DIAGRAM

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

5 SAMPLE CODE

5.1 NON-RECURSIVE CODE

5.1.1 Non-Recursive Code in C

```
int main(){
        int a = 2;
        int b = 3;
        int c;
        c = mult(2,3);
}

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

5.1.2 Non-Recursive Code in Assembly

```
.data
.byte num1 2
.byte num2 3
.text
la $sp num1
lw 0 $s1
lw 1 $s2
nand $ra $sp
sw 0 $s1
lw -1 $s1
addi -1 $sp
jal MULT
ADDITION: lw 1 $s2
add $s2 $s1
lw 0 $s2
MULT: lw -1 $ra
addi -1 $s2
sw 0 $s2
slt_0 $ra $s2
beq ADDITION
addi -1 $sp
sw 0 $s1
```

5.2 RECURSIVE CODE

5.2.1 Recursive Code in C

```
int main(){
        int d = 5;
```

```
        int e;
        e = sum(5);
}

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

5.2.2 Recursive Code in Assembly

```
function:
nand $s1, $s1
nand $s2, $s2
sl $s1, $s1
nand $s2 $s1
nand $s1, $s1   #$s1 = -2
sw 0, $s2   #save n in $s2 to memory

#then follows to sum (recursion)
sum:
add $sp, $s1    #adjust stack for 2 items $s1 = -2
sw 1, $ra   #save return address to memory
sw 2, $s2   #save current n to memory
nand $s1, $s1   #s1 becomes 1
sw 3, $s1   #saves $s1 (= 1)
slt_0 $s1, $s2    #slt_0=n > 1?, 0:1
slt_1 $s2, $s1    #slt_1=n>1?, 0:1
beq END     #if n < 1, jump to L1

nand $s1, $s1   #$s1 = -2
lw  3, $s2    #load 1 into $s2
add $s1, $s2    #$s1 = -1
lw 2, $s2   #load n into $s2
add $s2, $s1    #decrement n by 1
jal sum     #recursive call

lw 3, $s1
add $sp, $s1    #pop 2 items from stack (s1 needs to be 1)
add $sp, $s1
lw 1, $ra   #restore return address
lw 0, $s1   #restore original n to s1
add $s2, $s1    #s2 = s2 + s1
jr $ra      #return to calling address
```

```
END:
jr $ra        #return 0 if n is not > 1
```

6 MODULE I/O