Aardvark - 8-bit Processor

JEONG Andy, MACSHANE Gordon, SO Brenda

Contents

1 HARDWARE OVERVIEW

We are designing an 8-bit processor. Each instruction is 1 byte long. 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). $slt_0 and $slt_1 cannot be addressed directly, but rather utilized for jumping purposes. All registers are initialized at 0, while the $sp pointer is initialized at FF, which is the top of the memory stack.

2 MEMORY IMPLEMENTATION

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). We decided to have a 64 bytes of instruction because when we look at our largest program, it has a total of 28 lines. The .data section is limited to only four global bytes because . Due to the existence of a pseudo instruction -load address(la). If the programmer declared a global variable in the .data section, 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.

(Maybe we are going to implement cache?)

3

4 CORE INSTRUCTION SET

NAME,MNEUMONIC		FORMAT	OPERATIONS	OPCODE	FUNCT
ADD	add	R	R[rs] = R[rs] + R[rt]	000	1
NAND	nand	R	R[rs] = 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+2; PC = PC+immediate	111	n/a
JUMP REGISTER	jr	JR	PC = R[ra]	*101	1

* jump register can only jump to R[ra] and has the instruction as 00010011.

5 BASIC INSTRUCTION FORMAT

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

6 COMPILER

7 SAMPLE CODE

To demonstrate the functionality of the instruction set, three sets of codes were written. The first piece of code demonstrates basic arithmetic operations, including addition, subtraction, multiplication and division. The second and third sets of code demonstrates the summation from 1 to 10. The former performs the procedure recursively, the second performs the procedure non-recursively. Complementary C code is also provided for comparison.

7.1 ARITHMETIC CODE

7.1.1 Arithmetic Code in C

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

7.1.2 Arithmetic Code in Assembly

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

```
#a = $s1, b = $s2

        sw 0, $ra
        sw 1, $ra
        sw 2, $s2
        sw 3, $s3

subtraction:
        lw 3, $s1
        nand $s2, $s2
        addi 1, $s2
        add $s1, $s2
        sw 3, $s1

division:
        lw 3, $s2 #check if it has subtracted enough times
        slt_0 $ra, $s2 # 0 < sum
        lw 1, $s1
        addi 1, $s1
        sw 1, $s1
        lw 2, $s2 #put b into s2 for subtraction
        beq subtraction
        lw 1, $s1
        addi -1, $s1 #it always overshoots by one
        sw 1, $s1
        lw 1, $ra
```

7.2 NON-RECURSIVE CODE

7.2.1 Non-Recursive Code in C

```c
function()
{
  int i, sum;
  int n = 10;
  for (i = 0; i < n; i++)
  {
    sum += i;
  }
}
```

7.2.2 Non-Recursive Code in Assembly

```
function:

nand $s1, $s1
nand $s2, $s2
sl $s1, $s1
nand $s2 $s1     # i($s2) = 1
```

```
sw 0, $s2          # store i

nand $s1, $s1      #s1 becomes 1
sr $s1, $s1        #initialize sum($s1) = 0
sw 1, $s1          #store sum($s1) into offset 1 of $sp

add $s1, $s2        #n = 1
sl $s1, $s1
sl $s1, $s1
sl $s1, $s1         #n = 2^3 * n
add $s1, $s2        #n = 9
add $s1, $s2        #n($s1) = 10
sw 2, $s1          #save 10 to memory

nand $s1, $s1
nand $s2, $s2
sl $s1, $s1
nand $s1 $s2                # $s1 = 1
sw 3, $s1                   #store value 1

LOOP:
        lw 2, $s1           #load n
        slt_0 $s2, $s1      # i < n
        slt_1 $s1, $s2      # i > n
        beq END            #if i = n, go to END
        lw 0, $s2          #load i
        lw 1, $s1          #load sum
        add $s1, $s2       #sum($s1) += i
        sw 1, $s1
        lw 3, $s1
        lw 0, $s2          #load i to $s2
        add $s2, $s1       # i = i + 1
        jal LOOP
END:
        lw 1, $s1
```

7.3 RECURSIVE CODE

7.3.1 Recursive Code in C

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

7.3.2 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
```