S.P.utnik

Michael Hein
Brian Hulette
Erik Speyer
Mark Swanson

November 16, 2008

Contents

1 Introduction

Stack Processor utnik (S.P.utnik) is a 16-bit, stack-based processor. It is different from typical stack-based processors in a few ways:

- Arguments and Returns are stored in a special purpose register bank instead of being pushed onto the stack. This allows for quicker function calling (see “call” and “jret” RTL statements).
- Memory is byte-addressable, but it stores 2 byte words. This means instructions are a little quirky. Any instruction that does not require an immediate is just a 1 byte op-code. Instructions with immediate are a 1 byte op-code with the full 2 byte immediate tacked on. The long op-codes allow for more freedom in our instruction set and the full-length immediates allow for no limitatations for push and jumps.

2 Instruction Set

Instruction	Op-Code	Description
Arithmetic:		
add	04	Pop the top two items and push their sum
sub	0C	Pop the top two items and push next minus top
and	05	Pop the top two items and push next and top
or	01	Pop the top two items and push next or top
xor	02	Pop the top two items and push next xor top
not	03	Replace top with not top
shl	85XX	Bitwise-Shift top of stack X left
shr	8DXX	Bitwise-Shift top of stack X right
Stack Operations:		
push	C6XXXX	Push a 16-bit immediate
pop	07	Pop off the top of the stack
dup	08	Duplicate the top of the stack
swap	09	Swaps tops and next
Branches and Jumps:		
blt	CBXXXX	Branch if next <top
beq	CCXXXX	Branch if next = top
bne	CDXXXX	Branch if next != top
jump	CEXXXX	Jump to the 16-bit address in the immediate
call	CFXXXX	Jump to the 16-bit address in the immediate, write the return address to the top of the stack
jret	10	Jump to the address stored on top of the stack
Arguments and Returns:		
sarg	94XXXX	Set arg register X (0-3) to the value popped off the stack
garg	95XXXX	Get arg register X (0-3), push to stack
sret	96XXXX	Set ret register X (0-3) to the value popped off the stack
gret	97XXXX	Get ret register X (0-3), push to stack
Input and Output:		
dump	11	Dump to to the display register
ipsh	12	Pushes the input register value onto the stack
dpsh	14	Pushes the display register value onto the stack

2.1 Machine Language Conversion

Converting each instruction to machine language is relatively trivial. For each instruction just write the op-code, then whatever immediate is passed as a 2-byte value. The only odd thing is that the immediates of gret and sret must be increased by 4. This is because they are actually used as addresses to a register bank which stores boths arguments and returns, the arguments are the first 4 registers, and the returns are the last 4.

3 Programmer's Guide

We understand our instruction set has some quirks, so we've created this programmers guide to make it easier for new programmer's to use our assembly language.

3.1 Basic Operations

To push an immediate value onto the stack use "push":

```
push 3
push 8
```

Stack
8
3
...

To do basic arithmetic operations use add, sub, and, or, xor:

```
add
```

Stack
11
...

dup will copy the value on top of the stack and push it:

```
dup
```

Stack
11
11
...

3.2 Branching and Jumping

All branches and jumps have no limitations. S.P.utnik uses 16-bit immediate values, meaning branches and jumps can move anyhere in the program. a jump instruction unconditionally jumps to the address in the immediate:

```
        push    16
        jump    skip
        pop
skip:   push    32
```

Stack
32
16
...

Branches pop the two items that they compare.

```
        push    56
        dup
        beq     skip
        push    0
skip:   push    1
```

Stack
1
...

3.3 Register Conventions

The programmer is not allowed direct access to any registers, but they are allowed to modify arg and ret registers with sret and sarg instructions. Set argument registers to arguments for a procedure prior to calling. Procedure should set return registers to return values prior to returning.

3.4 Procedure Conventions

Anything stored in the argument or return registers which you intend to keep after a procedure call, you must store on the stack. Any called procedure absolutely must restore the stack to its original state before returning.

3.5 Putting it Together

The following is a sample procedure we wrote to demonstrate the use of our instruction set. It calculates the factorial of the number stored in arg0 and writes the result to ret0.

```
Fact:   garg    0     # Put arg0 on stack
        dup           # dup for comparison
        push 2     # if(n < 2)
        blt     Exit  # return n
        dup
        push    1
        sub           # push n-1
        sarg    0     # arg0 = n-1
        call    Fact
        gret    0     # push Fact(n-1)
        mul           # n * Fact(n-1)
Exit:   sret    0     # ret0 = n*Fact(n-1)
                      # OR 1 OR 0
        jret          # return
```

This is a good example of writing basic procedures with our instruction set, and it demonstrates that S.P.utnik is capable of running recursive procedures. It will not actually assemble because we decided not to implement multiply, but if you pretend that multuply is a procedure implemented somewhere else, it gets the point across. The following is the final program we are supposed to implement on our processor. It is supposed to find a relatively prime value of a number that the user inputs. The user inputs the number 4 bits at a time with the switches then pushes a button to run the algorithm. We have not actually successfully put the program on the processor yet, but I think the problem was with our assembling, and not the actual code.
It also demonstrates procedure calls in S.P.utnik. Note the use of argument and return registers when calling Euclid.

```
        wait          # wait for interrupt
        dup
        push    1     # if code is 1
        beq     inpt  # run 'inpt'
        push    0     # if code is 0
        beq     run   # run 'run'
run:    dpsh
        sarg    0
        call    Euclid
        gret    0
        dump
        jret

inpt:   ipsh          # push switches
        dpsh          # push display
        shl     4     # shift disp
        or            # and tack on switches
        dump          # write new display
        jret

Euclid: push    2     # m = 2
        sarg    1
Loop:   call    gcd   # call gcd(n, m)
        garg    0     # push n
        garg    1     # push m
        gret    4     # push return of gcd
        push    1     # top of stack = 1
        beq     exit  # m == 1 ? exit
        push    1     # top of stack = 1
        add           # m = m + 1
        sarg    1     # arg1 = m
        sarg    0     # arg0 = n
        jump    Loop
gcd:    garg    0     # push a
        sret    4     # ret0 = a
        garg    1     # push b
        sret    5     # ret1 = b
L1:     push    0
        gret    5     # push b
        bne     L2    # if (b > 0)
        jret          # return to Loop
L2:     gret    5     # push b
        gret    4     # push a
        blt     L3    # if (b > a)
        gret    5     # push b
        gret    4     # push a
        sub           # a-b
        sret    4     # a = a-b
        jump    L1
L3:     gret    4     # push a
        gret    5     # push b
        sret    4     # a = b
        sret    5     # b = a
        jump    L1
exit:   garg    1     # return
        sret    4     # relatively
        jret        # prime number
```

3.5.1 Working Programs

The next few programs were written by us to test our processor as we debugged it, all of them have run on the FPGA correctly. They have been provided here because they demonstrate basic concepts well:

```
        push     FFFF
        dump
```

This is a very simple program we wrote to test the display, It should just display FFFF if its working properly.

```
        push     000F
        push     0001
        add
        dump
```

This program will simply add F and 1, then display the result. It should display 0010 if its working properly.

```
        push     0001
        push     000F
        dup
        beq      skip
        not
skip:   dump
```

This program is meant to test branching, if the branch is not working properly, it should not the output before displaying it. If all instructions are working properly is should display 0001.

```
        push     0000
        sarg  0
        call     not
        gret     0
        dump
        wait
not:    garg     0
        not
        sret     0
        jret
```

This program is supposed to test procedure calling. If it works correctly it should disp not 0000 or FFFF.

```
        wait
        dup                # interrupt handling
        push     1
        beg      inpt
        push     0
        beq      run
run:    dpsh
        not
        dump
        jret
inpt:   ipsh
        dpsh
        shl      4
        or
        dump
        jret
```

This program is meant to be the final test before trying the final program. It is essentially the same, except it nots the input instead of calculating the gcd.

4 Component Specifications

4.1 Memory

We will use byte-addressable memory with 2 byte words. There are 3 inputs: a 16-bit address, a 16-bit data line, and a 1-bit active high write signal. There are 2 outputs: an 8-bit read signal which is the byte the address points to and a 16-bit read signal which is the addressed byte and the following byte.

4.2 Register Bank

Arguments and Returns are all stored in registers rather than pushed onto the stack. This allows for much simpler function calling and returning. The register bank is designed to store these values, it uses a 3 bit address for 8 registers (4 arguments and 4 returns), a 16-bit data input and a 1-bit active high write signal. The output is written to a 16-bit reaad signal.

4.3 ALU

The ALU takes 2 16-bit operands, A and B, as inputs and a 4-bit ALUOp to decide which operation to run. It performs basic math operations like add, subtract, and, or and xor using 1-bit ALUs connected in a ripple-carry configuration. It also contains a barrel shifter to shift the A input B bits to the right or left.

4.4 Special Purpose Registers

There are several other special purpose registers which store intermediate data. They are all 16-bit with active-high write signals unless otherwise noted.

4.4.1 Function Pointer

The function pointer, or fptr, stores the address of the current instruction or immediate in memory.

4.4.2 Stack Pointer

The stack pointer, or sptr, stores the address of the top of the stack in memory.

4.4.3 ALUOut

ALUOut stores temporary results from the ALU until they can be written to memory or another register.

4.4.4 A and B

A and B store values that are read from memory.

4.4.5 Instruction Register

The instruction register, IR, is an 8-bit register that stores the op-code of the instruction after it is read from memory. It's ouput is constantly fed to control.

5 Contoller Specifications

The controller has many signals which it uses to control the various operations of the processor. These are listed below.

spWrite One bit signal for writing the stack pointer register

fpWrite One bit signal for writing the function pointer

aWrite One bit signal to control writing register A

bWrite One bit signal to control writing register B

irWrite One bit signal to control writing to the instruction register

aluOutWrite One bit signal to control writing to the aluOut register

memWrite One bit signal for writing to the memory

memData Four bit signal to control what gets written to memory

memAddr 1 bit signal to control the mux that decides what address is being accessed in memory

fpData Two bit signal to control what gets written to the function pointer mux

aluSrcA Two bit signal to control the mux on the A side of the alu

aluSrcB Two bit signal to control the mux on the B side of the alu

regWrite One bit signal to control writing to the register file

aluOp 4 bit signal to control the mode of operation for the alu

dispWrite One bit signal to control writing to the display

blt, beq, bne 1 bit signals that control branching

6 RTL Statements

6.1 Arithmetic

6.1.1 add/sub/and/or/xor

IR = MEM8[fptr]
fptr = fptr + 1

A = MEM16[sptr]
sptr = sptr + 2

B = MEM16[sptr]
ALUOut = A op B
MEM16[sptr] = ALUOut

6.1.2 shl/shr

R ret0 = 1 OR ret0 = 0 IR = MEM8[fptr] fptr = fptr + 1
A = MEM16[sptr] sptr = sptr + 2
B = MEM16[sptr] fptr = fptr + 2
ALUOut = A <<OR >>B
sptr = sptr - 2
MEM16[sptr] = ALUOut

6.1.3 not

IR = MEM8[fptr] fptr = fptr + 1
A = MEM16[sptr] sptr = sptr + 2
A = MEM16[sptr]
MEM16[sptr] = !A

6.2 Stack Operations

6.2.1 push

IR = MEM8[fptr] fptr = fptr + 1
sptr = sptr + 2 A = MEM16[sptr]
fptr = fptr + 2 A = MEM16[fptr]
sptr = sptr - 4
MEM16[sptr] = A

6.2.2 pop

IR = MEM8[fptr] fptr = fptr + 1
sptr = sptr + 2 A = MEM16[sptr]

6.2.3 dup

IR = MEM8[fptr] fptr = fptr + 1
sptr = sptr + 2 A = MEM16[sptr]
sptr = sptr - 4
MEM16[sptr] = A

6.2.4 swap

IR = MEM8[fptr] fptr = fptr + 1
sptr = sptr + 2 A = MEM16[sptr]
B = MEM16[sptr]
sptr = sptr - 2 MEM16[sptr] = A
MEM16[sptr] = B

6.3 Branches and Jumps

6.3.1 blt/beq/bne

IR = MEM8[fptr] fptr = fptr + 1
sptr = sptr + 2 A = MEM16[sptr]
B = MEM16[sptr] sptr = sptr + 2
if((A-B) op 0) then fptr = MEM16[fptr]
else fptr = fptr + 2

op is <, =, or != depending on the instruction

6.3.2 jump

IR = MEM8[fptr] fptr = fptr + 1
sptr = sptr + 2 A = MEM16[sptr]
sptr = sptr - 2 A = MEM16[fptr]
fptr = A

6.3.3 call

IR = MEM8[fptr] fptr = fptr + 1
sptr = sptr + 2 A = MEM16[sptr]
sptr = sptr - 4 A = MEM16[fptr]
ALUOut = fptr + 2 fptr = A
MEM16[sptr] = ALUOut

6.3.4 jret

IR = MEM8[fptr]
fptr = fptr + 1
sptr = sptr + 2
A = MEM16[sptr]
fptr = A

6.4 Arguments and Returns

6.4.1 sarg/sret

IR = MEM8[fptr]
fptr = fptr + 1
sptr = sptr + 2
A = MEM16[sptr]
B = MEM16[fptr]
fptr = fptr + 2
REG[B] = A

6.4.2 garg/gret

IR = MEM8[fptr]
fptr = fptr + 1
sptr = sptr + 2
A = MEM16[sptr]
B = MEM16[fptr]
sptr = sptr - 4
MEM16[sptr] = REG[B]
fptr = fptr + 2

6.5 Input and Output

6.5.1 ipsh

IR = MEM8[fptr]
fptr = fptr + 1
sptr = sptr + 2
A = MEM16[sptr]
sptr = sptr - 4
MEM16[sptr] = INPT

6.5.2 dpsh

IR = MEM8[fptr]
fptr = fptr + 1
sptr = sptr + 2
A = MEM16[sptr]
sptr = sptr - 4
MEM16[sptr] = DISP

6.5.3 dump

IR = MEM8[fptr]
fptr = fptr + 1
sptr = sptr + 2
A = MEM16[sptr]
sptr = sptr - 2
DISP = A

7 Building the Processor

The datapath of any processor is a very complicated machine, and S.P.utnik is no exception. This means that there are plenty of mistakes to be made when building the processor. To keep this from happening we have divided up the work into several testable steps that will allow us to catch mistakes earlier.

7.1 Component Testing

First we need to build all of the individual components. We've thought up methods for testing each as they are complete.

7.1.1 Memory

Simply write data to memory in various locations, including the very first word and the very last word, then read back each value that was written. Then ensure that every value read can be explained by what was written.

7.1.2 Register Bank

Testing for the register bank will be very similar. However, there are far fewer location in the register bank so we can test each exhaustively.

7.1.3 ALU

Every ALU operation must be tested with several different inputs to ensure that they work correctly. Also, the zero output can be checked by subtrating two equal values, and the slt value can be tested by subtracting a large value from a smaller one.

7.1.4 Special Purpose Registers

Testing the special purpose registers is very simple, simply write a few values, and ensure that it outputs them correctly.

7.2 Integration Testing

7.2.1 Step 1: ALU, A and B Registers

First we will add the ALU, since it is the core of the processor. However, its useless without operands, so we will add the A and B registers and the ALUSrc muxes hard-wired to choose A and B. We can then set the input values to A and B and watch the ALU output to make sure they work together correctly.

7.2.2 Step 2: ALUOut, Constants

We will add the ALUOut register and connect its data input to the ALU output. Hopefully, we will be able to watch the results get written to it. We will also add the remaining connections to the ALUSrcB register (all constants).

7.2.3 Step 3: Pointers

Now we can add the fptr and sptr registers, which contain the address of the current instruction and the top of the stack, respectively. We should be able to write values to them that come from the ALU output or register A.

7.2.4 Step 4: Register Bank

We'll add the Register bank, which is used to store arguments and returns. We will just need to check that it can write a value stored in A to the address stored in B.

7.2.5 Step 5: Memory and IR

Next we can add the final datapath components: the memory and the instruction register. To test this we can read various values from memory and write them into A,B, or IR.

7.2.6 Step 6: Control

Testing control will be a significant step. We will essentially need to load a program into memory, and observe how well it executes. If there are problems, we will know that they arose from control because the datapatch has been thoroughly tested.

8 Processor Performance

We are able to evaluate our processor at many levels using the Xilinx software, the following is the data we gathered from it:

Instructions for Euclid's Algorithm 34

Cycle Time 66.830 ns

Gate Count 76,622

Slice flip-flops 4%

4-input LUTs 11%

Occupied Slices 20%

Total 4-input LUTS 17%

Bonded IOBs 23%

GCLKs 4%