ISA Alpha Release
Marc Slaughter and Nicholas Otto
February 22, 2013

1 ISA specification

We have broken our ISA into two different operation types, similar to "R" and "I" type for MIPS. We have "Normal" and "S" type operations.

1.1 Normal Type Operations

Asembly format: opcode operand operand operand where operand is a register, or an offset, depending on the opcode. We have a 4bit op code and 3 bits for each operand Thus, there is a possibility for 16 operations, and 8 general registers (the S type operations access special registers).

Operations:

1.
```
mul $dest, $gen1, $gen2
0001 *** *** ***
    does what you think
    returns lower 34 bits into $dest, higher 34 bits are ignored
    $dest = lower34($gen1 * $gen2)
```

2.
```
add $dest, $gen1, $gen2
0010 *** *** ***
    does what you think
    ignore overflow
    $dest = $gen1 + $gen2
```

3.
```
sub $dest, $gen1, $gen2
0011 *** *** ***
    does what you think
    ignore overflow
    $dest = $gen1 - $gen2
```

4.
```
or  $dest, $gen1, $gen2
0100 *** *** ***
    logical or
```

5.
```
nor  $dest, $gen1, $gen2
0101 *** *** ***
     ~($gen1 or $gen2)
```

6. `sr  $dest, $gen1, $gen2`

```
0110 *** *** ***
    bitwise shift-right
    $dest = $gen1 >> $gen2
```

7. `lw  $value, $base, $offset`

```
0111 *** *** ***
    load word (34 bits).
    $value = M[$base + $offset]
```

8. `sw  $value, $base, $offset`

```
1000 *** *** ***
    store word (34 bits)
    M[$base + $offset] = $value
```

9. `bne $gen1, $gen2, PC_offset`

```
1001 *** *** ***
    branch not equal
    if($gen1 != $gen2):
        PC = PC+InstL+(InstL)*PC_offset
    note: this will be handled by assembler so we can implement label functionality
```

10. `beq $gen1, $gen2, PC_offset`

```
1010 *** *** ***
    branch equal
    if($gen1 == $gen2):
        PC = PC+InstL+(InstL)*PC_offset
    note: this will be handled by assembler so we can implement label functionality
```

11. `slt $dest, $gen1, $gen2`

```
1011 *** *** ***
    set if less than
    if($gen1 < $gen2):
        $dest = 1
    else:
        $dest = 0
```

1.2 Special Type Operations

Assembly format: opcode operand1 operand2 where opcode is a 4 bit op code, operand1 is 3 bit specifying a value, and operand2 is a 6 bit special designator (which in some cases is a general register).

12. `j   $op1, $op2`

```
1100 *** ******
    jump to PC + instruction length (13 bits) + $op1+$op2 instructions
    PC = PC+InstL+ InstL*($op1+$op2)
```

```
j label
    Will be converted by the assembler the correct format
```

13. set $dest, immediate

```
1101 *** *** ***
        $dest = 0 | immediate
```

14. sl $op1, immediate

```
1110 *** *** ***
        $op1 = $op1 << immediate
```

15. TRAP op1, op2

```
1111 *** *** ***
        A "catch-all" operation, where the function to execute is specified by the immediate op1.
```

(a) QUIT

```
1111 000 ******
        return op2 = $gen and exit program
```

(b) IN

```
1111 001 ******
        store into $gen0 from $op2 (where $op2 is the $channel)
```

(c) OUT

```
1111 010 ******
        send $0 to $op2 (where $op2 is the $channel)
```

(d) function call

```
1111 011 ******
        op2 is a label specifying the next instruction to execute (using label functionality)
        1. increment $fp by 34
        2. store PC+13 -> M[$fp]
        3. PC = label (jump)
```

(e) function return

```
1111 100 ******
        op2 is not needed
        1. PC = M[$fp]  (jump back)
        2. decrement $fp by 34
```

(f) stack push

```
1111 101 ******
        op2 = $gen register to push onto stack
        $sp = $sp+34
        M[$sp] = $op2
```

(g) stack pop
```
1111 110 ******
        op2 = destination register $gen0-$gen7
        $op2 = M[$sp]
        $sp = $sp-34
```

REGISTERS

```
$gen0-$gen7     general registers
$sp             stack pointer register (for trap 5,6)
$fp             function pointer register (for trap 3,4)
$ch0 - $ch15    channel registers (for trap 1,2)
```

```
PC = program counter
```

When specifying immediates, the assembler will accept any format (and take only the appropriate size). So

```
b1001 = 0x9 = 9.
```

2 Our ISA implementation of UltraMega

```
#define
pc = $1
mem = $2
inst = $3
op = $4
srcA = $5
srcB = $6
dest = $7
temp = $0
INT_SIZE = 1
#end


tagA: add inst, mem, pc
        lw op, inst // Get op
        lw srcA, 1(inst) // Get srcA
        lw srcB, 2(inst) // Get srcB
        lw dest, 3(inst) // Get dest
        set temp, 1
        add pc, pc, temp

********case 0**********************
        set temp, 0
        bne op, temp, tag1
```

```
        add srcA, mem, srcA
        lw srcA, srcA //srcA = M[srcA]
        add srcB, mem, srcB
        lw srcB, srcB //srcB = M[srcB]

        sub temp, srcA, srcB
        add dest, mem, dest
        sw temp, dest
        j tagA


********case 1**********************
tag1: set temp, 1
        bne op, temp, tag2
        add srcA, mem, srcA
        lw srcA, srcA //srcA = M[srcA]
        srl srcA, srcA, 1

        add dest, mem, dest
        sw srcA, dest //store
        j tagA

********case 2**********************
tag2: set temp, 2
        bne op, temp, tag3
        add srcA, mem, srcA
        lw srcA, srcA //srcA = M[srcA]
        add srcB, mem, srcB
        lw srcB, srcB //srcB = M[srcB]
        nor temp, srcA, srcB

        add dest, mem, dest
        sw temp, dest //store
        j tagA

********case 3**********************
tag3: set temp, 3
        bne op, temp, tag4
        add srcA, mem, srcA
        lw srcA, srcA //srcA = M[srcA]

        add temp, mem, srcA
        lw srcA, temp //srcA = M[M[srcA]]

        add srcB, mem, srcB
        lw srcB, srcB //srcB = M[srcB]
        add dest, mem, dest

        sw dest, srcA // dest = MMsrcA
        sw srcA, srcB // MMsrcA = MsrcB
        j tagA
```

```
********case 4*********************
tag4: set temp, 4
        bne op, temp, tag5
        add srcA, mem, srcA
        lw srcA, srcA
        add temp, mem, dest
        lw dest, temp //dest = M[dest]
        IN dest, srcA
        j tagA

********case 5*********************
tag5: set temp, 5
        bne op, temp, tag6
        add temp, mem, srcA
        lw srcA, temp

        mul srcB, srcB, INT_SIZE
        add temp, mem, srcB
        lw srcB, temp //srcB = M[srcB]
        OUT srcA, srcB
        j tagA

********case 6*********************
tag6: set temp, 6
       bne op, temp, tag7
        add temp, mem, srcA
        lw srcA, temp
        add temp, mem, srcB
        lw srcB, temp

        add temp, mem, dest
        lw dest, temp
        sw dest, pc
        set temp, 0
        slt srcA, srcA, temp
        beq srcA, temp, tagA

        set pc, srcB
        j tagA

********case 7*********************
tag7: QUIT pc
```

3 Fibonacci Implementation

```
% Fibonacci implementation.  "n" is the top of the $sp
```

```
:fibo
    TRAP    6, $1       //put n in $1
    set     $2, 0
    slt     $3, $1, $2  // $3 = if(n < 0)
    set     $2, 1
    beq     $2, $3, label1

    set     $2, 3
    slt     $3, $1, $2  // $3 = if(n < 3)
    set     $2, 1
    beq     $2, $3, label2
    //else if (n=29)
    set     $2, 29
    beq     $1, $2, label3

    set     $4, 1 //$3 is running total
    sub     $2, $1, $4  // $2 = n-1
    sub     $3, $2, $4  // $3 = n-2
    TRAP    5, $2       push n-1
    TRAP    5, $3       push n-2
    TRAP    3, fibo     call fibo
    TRAP    6, $1       $1 = pop
    TRAP    6, $2       $2 = pop
    TRAP    5, $1       push temp1
    TRAP    5, $2       push temp2
    TRAP    3, fibo     call fibo
    TRAP    6, $1       pop
    TRAP    6, $2       pop
    add     $0, $1, $2  add
    TRAP    5, $0       push
    TRAP    4, 0

label1:
    set     $1, b0
    set     $2, b1
    sub     $0, $1, $2  // -1   = b1111111111111111111
    TRAP    5, $0
    TRAP    4, 0

label2:  // 1    = b1
    set     $0, b1
    TRAP    5, $0
    TRAP    4, 0

label3:  // 514229 = b1111101100010110101
    set     $0, 0
    set     $1, b111110
    sl      $1, 13
    or      $0, $1, $0
    set     $1, b110001
    sl      $1, 7
```

```
    or      $0, $1, $0
    set     $1, b011010
    sl      $1, 1
    or      $0, $1, $0
    set     $1, b1
    add     $0, $0, $1
    TRAP    5, $0
    TRAP    4, 0

label4  // 832030 = b110010 110010 000111 10
    set     $0, b0
    set     $1, b110010
    sl      $1, 14
    or      $0, $1, $0
    set     $1, b110010
    sl      $1, 8
    or      $0, $1, $0
    set     $1, b000111
    sl      $1, 2
    or      $0, $1, $0
    set     $1, b10
    add     $0, $0, $1
    TRAP    5, $0
    TRAP    4, 0


label5  // 4807526976 = b100011 110100 011010 000101 001000 000
    set     $0, 0
    set     $1, b100011
    sl      $1, 27
    or      $0, $1, $0
    set     $1, b110100
    sl      $1, 21
    or      $0, $1, $0
    set     $1, b011010
    sl      $1, 15
    or      $0, $1, $0
    set     $1, b000101
    sl      $1, 9
    or      $0, $1, $0
    set     $1, b001000
    sl      $1, 3
    or      $0, $1, $0
    TRAP    5, $0
    TRAP    4, 0



label6: ;7778742049 =  b111001 111101 001100 010111 100100 001
    set     $0, 0
    set     $1, b111001
```

```
sl      $1, 27
or      $0, $1, $0
set     $1, b111101
sl      $1, 21
or      $0, $1, $0
set     $1, b001100
sl      $1, 15
or      $0, $1, $0
set     $1, b010111
sl      $1, 9
or      $0, $1, $0
set     $1, b100100
sl      $1, 3
or      $0, $1, $0
set     $1, b1
add     $0, $1
TRAP    5, $0
TRAP    4, 0
```

4 Function call ABI

Function calls are implemented using stacks. When making a function call

```
TRAP    3, label
```

the value of the program counter, PC, plus 13 is placed onto the function stack (indicated by $fp). $fp is then incremented to point to the next open stack opsition. PC is then given the value associated with the label. A call to return will set PC to the value popped off the function pointer stack.

If the user would like to pass values, they must first push those values onto the general value stack, whose entry point is indicated by $sp. The called function must then pop the values it needs off of the stack. With this method, the general registers can be overridden without risking overriding parameters.

5 Evaluation of EDP for MIPS and our ISA

The Ultramega code did not change significanly, so the EDP evaluation remains the same.

6 Comments

Our original implementation was general enough to not need much modifying except to handle function calls. We were able to convert IN, OUT, QUIT, into a general TRAP function, which allowed for adding the shift left instruction, to allow for easier adding of static numbers. Thus, the implementation of fibonacci only required adding the function call functionality.