BAEJ

BAEJ is a Reduced Instruction Set Computer Architecture which implements a load store architecture

Registers

Registers	Address	Use
.f0 - .f15	0-15	General purpose 'function registers' where data is not lost after a function call
.t0 - .t28	16-44	General purpose 'temporary registers' where data may be overridden during a function call
.a0 - .a5	45-50	Argument registers for function calls
.m0 - .m5	51-56	Accumulator register on which default mathematical operations are committed
.cr	57	Compiler register
.pc	58	Program counter register
.v0 - .v1	59-60	Return value register from a function call
.ra	61	Return address register
.sp	62	Stack pointer register
.z0	63	Register always holding the value 0

Function Registers

Function registers are magic! You are able to use them (.f0 - .f15) without worry of loosing the data after a function call. The best part? You don't even need to back them up to a stack... or anywhere! Thanks to some nifty magic. **How does the magic work?** The 16 function registers available to you anywhere are actually part of a larger register file consisting of 128 registers. To address all 128 of these registers we have designed a 3 bit register that is called the function call counter (FCC). The FCC stores the count of how many function calls have been made without returning. By using the following formula we can get the next 16 registers in the larger file.

```
target = (FCC * 16) + register
```

For example, if we have made only one function call, FCC would be 1. If we wanted to address our third function register (.f2) we would take `1 * 16 + 2` to get 18. This means we would be addressing the 18th register in the larger file by using .f2 in our function. This makes sense because registers 0 - 15 would be owned by the place from where our function would be called, thus the 18th register is the third one available to us. If more than 7 function calls are made (the largest value the FCC can hold) then what?

> **Options** * Use special memory unit for both memory and backing up registers (Hard but extra credit) * Increase FCC to 12 bits and limit to 4096 function calls (easy) * Use a dedicated memory unit to backup entire register file (medium possibly extra credit) 128 bit words

Contingency

If we continue to run into issues with the proposed plan for f registers as described above, or hit an impassable roadblock, our contingency plan is to only use 15 registers addressed normally which will be required by functions to be backed up on the stack in memory.

Machine Code Formats

I |15 OPCODE 12|11 RS 6|5 RD 0| (1st word)

|15 IMMEDIATE 0| (2nd word)

I type instructions use the format above. They are multi-word instructions with the first word consisting of a 4 bit op code followed by two 6 bit register addresses. The second word will be the 16 bit immediate value used in the instruction.

G |15 OPCODE 12|11 RS 6|5 RD 0| (1st word)

G type instructions use the same format as I type as described above. They do not, however, have an immediate and only have one word in their machine code format.

Instructions

Instruction	Type	OP	Usage	Description	Rtl
`lda`	I Type	0000	`lda .rs[index] .rd`	Loads a value from memory to rd	`rd=Mem[rs+index]`
`ldi`	I Type	0001	`ldi .rd immediate`	Loads an immediate to rd	`rd=immediate`
`str`	I Type	0010	`str .rs[index] .rd`	Stores value in rd to memory	`Mem[rs+index]=rd`
`bop`	I Type	0011	`bop immediate`	Changes pc to immediate	`pc=immediate`
`cal`	I Type	0100	`cal immediate`	Changes pc to immediate and sets a return address	`ra=pc+4pc=immediate`
`beq`	I Type	0101	`beq .rs .rd immediate`	Changes pc to immediate if rs and rd are equal	`if rs==rd pc=immediate;`
`bne`	I Type	0110	`bne .rs .rd immediate`	Changes pc to immediate if rs and rd aren't equal	`if rs!=rd pc=immediate;`
`sft`	I Type	0111	`sft .rs .rd immediate`	Shifts value in rs to rd by immediate. Positive shifts left, negative shifts right	`rd=rs<<immediate`
`cop`	G Type	1000	`cop .rs .rd`	Copies the value of rs to rd while retaining the original value of rs	`rd=rs`
`slt`	G Type	1010	`slt .rs .rd`	Sets cr to a value other than 0 if rs is less than rd	`cr=rs<rd?1:0`
`ret`	G Type	1011	`ret`	Sets pc to the value in ra	`pc=ra`
`add`	G Type	1100	`add .rs [.rm]`	Adds rs into the accumulator*	`[rm]+=rs`
`sub`	G Type	1101	`sub .rs [.rm]`	Subtracts rs from the accumulator*	`[rm]-=rs`
`and`	G Type	1110	`and .rs [.rm]`	Ands rs with the accumulator*	`[rm]^=rs`
`orr`	G Type	1111	`orr .rs [.rm]`	Ors rs with the accumulator*	`[rm]|=rs`

***optional argument of .rm specifies an accumulator register to operate on (defaults to .m0)**

Function Calls

Function calls are made easy with BAEJ. When calling a function the programmer simply places the arguments in registers a0 - a5 and uses the command `cal <FUNCTION>`. The instruction will jump the program counter to the address of the function while also putting the previous value of the program counter plus 2 into the return address register. The function will then return with `ret` which returns to the address in the ra register. The programmer can expect their data in f registers to be retained while they should not expect data in any other register to be retained. After a function returns, returned values will be in the v registers.

Examples

Common Assembly/Machine Language Fragments

Loading an address into a register

BAEJ Code

```
ldi .f0 addr
lda .f0[0] .f1
```

Machine Code Translation (assuming the value stored in addr is 280)

```
0x00            0001 000000 000000
0x02            0000 000100 011000
0x04            0000 000000 000001
0x06            0000 000000 000000
```

Sum Values from x (a0) to y (a1) assuming x < y

BAEJ Code

```
        cop .a0 .m0
        cop .a0 .m1
        ldi .f0 1
loop:   add .f0 .m1
        add .m1
        slt .m1 .a1
        bne .z0 .cr loop
```

Machine Code Translation (Assuming the address of loop is 0x8)

```
0x00            1000 101101 110011
0x02            1000 101101 110100
0x04            0001 000000 000000
0x06            0000 000000 000001
0x08    loop:   1100 000000 110100
0x0A            1100 110100 110011
0x0C            1010 110100 101110
0x0E            0110 111111 111001
0x00            0000 000000 001000
```

Modulus

BAEJ Code

```
loop:   add .a1
        slt .a0 .m0
```

```
        bne .z0 .cr loop
        sub .a1
        cop .a0 .m1
        sub .m0 .m1
```

Machine Language Translation (Assuming the address of loop is at 0x0)

```
0x00    loop:   1100 101110 110011
0x02            1010 101101 110011
0x04            0110 111111 111001
0x06            0000 000000 000000
0x08            1101 101110 110011
0x0A            1000 101101 110011
0x0C            1101 110011 110100
```

Euclid's Algorithm

C Code

```
// Find m that is relatively prime to n.
int
relPrime(int n)
{
   int m;

   m = 2;

   while (gcd(n, m) != 1) {  // n is the input from the outside world
     m = m + 1;
   }

   return m;
}

// The following method determines the Greatest Common Divisor of a and b
// using Euclid's algorithm.
int
gcd(int a, int b)
{
  if (a == 0) {
    return b;
  }

  while (b != 0) {
    if (a > b) {
      a = a - b;
    } else {
      b = b - a;
    }
  }

  return a;
}
```

BAEJ Translation

```
# Greatest common divisor
gcd:    bne .a0 .z0 cont
        cop .a1 .v0
        ret

        cop .a0 .m0         # Copy arguments into accumulators
        cop .a1 .m1

cont:   beq .m1 .z0 end     # While b != 0
        slt .m1 .m0
        beq .cr .z0 else    # If a > b
        sub .m1
        bop cont

else:   sub .m0 .m1         # Else
        bop cont

end:    cop .m0 .v0
        ret

# Relative prime
relP:   ldi .m0 2           # .m0 stores value of m
                            # .a0 stores value of n
loop:   cop .m0 .a1
        cal gcd
        ldi .t1 1
        beq .v0 .t1 done    # While gcd(n,m) != 1
        add .t1             # m = m + 1
        bop loop

done:   cop .m0 .v0
        ret                 # return m
```

Machine Code Translation

```
0x00    relP:   0001 110011 000000
0x02            0000 000000 000010
0x04    loop:   1000 110011 101110
0x06            0100 000000 000000
0x08            0000 000000 011100  # address[gcd]
0x0A            0001 010001 000000
0x0C            0000 000000 000001
0x0E            0101 111011 010001
0x10            0000 000000 011000  # address[done]
0x12            1100 010001 000000
0x14            0011 000000 000000
0x16            0000 000000 000100  # address[loop]
0x18    done:   1000 110011 111011
0x1A            1011 000000 000000
0x1C    gcd:    0110 101101 111111
0x1E            0000 000000 101000  # address[cont]
0x20            1000 101110 111011
0x22            1011 000000 000000
0x24            1000 101101 110011
```

```
0x26            1000 101110 110100
0x28    cont:   0101 110100 111111
0x2A            0000 000000 111110  # address[end]
0x2C            1010 110100 110011
0x2E            0101 111010 111111
0x30            0000 000000 111000  # address[else]
0x32            1101 110100 000000
0x34            0011 000000 000000
0x36            0000 000000 101000  # address[cont]
0x38    else:   1101 110011 110100
0x3A            0011 000000 000000
0x3C            0000 000000 101000  # address[cont]
0x3E    end:    1000 101101 111011
0x40            1011 000000 000000
```

RTL

I Types

`lda`	`str`	`ldi`	`beq/bne`	`sft`	`bop`	`cal`
IR = Mem[PC] ImR = Mem[PC+2] PC += 2						
PC += 2 A = Reg[IR[11:6]] B = Reg[IR[5:0]]					PC = ImR	ra = PC + 2 PC = ImR
ALUout = A + ImR		Reg[IR[5:0]] = ImR	if(A==B) PC = ImR	ALUout = A << ImR		Fcache[FCC] = Reg[15:0] FCC += 1
Memout = Mem[ALUout]	Mem[ALUout] = B			Reg[IR[5:0]] = ALUout		
Reg[IR[5:0]] = Memout						

G Types

`cop`	`slt`	`Other G Types`	`ret`
IR = Mem[PC] PC += 2			
A = Reg[IR[11:6]] B = Reg[IR[5:0]]			PC = ra FCC -= 1
Reg[IR[5:0]] = A	AlessThanB = A < B ? 1 : 0	ALUout = A op B	Reg[15:0] = Fcache[FCC]
	cr = ALessThan	Reg[IR[5:0]] = ALUout	

Hardware Components

- PC Adder

  This component is a combinational logic component which takes two inputs, one 16 bit input, and one 1 bit input (the first wire from the op code). With the single bit input low, the adder will add 2 to the 16 bit input. With the single bit input high, the adder will add nothing to the 16 bit input. This is the component that will be used to increment the adder.

- Address Adder

  This component is a standard adder which will be used to add the A and ImR registers when addressing memory for load and store instructions. It will have the two 16 bit inputs and one 16 bit output and no control signals

- Registers

  A general register component takes in one 16 bit input for writing a value into the register, and has one 16 bit output signal for reading the value out of the register. Registers do not have any control signals. A general register component implements the following RTL symbols:

Registers	Description
A & B	These registers are used for holding data values retrieved out of the register file for use as inputs in the ALU.
ALUout	This register is used for holding the output of the ALU (for lda and ldi).
FCC	The FCC (function call counter) register is used to hold the current count of function calls for use in addressing f-register backups to the f-register cache.
PC	The PC (program counter) register is used to track the current address in memory of a program.
ra	The ra (return address) register gets the address of PC+2 when a function call is made.
cr	The cr (compiler register) is for temporary use by the assembler in functions such as slt.
IR	The IR (instruction register) holds the 16 bits of instruction pulled from instruction memory at the address value stored in PC.
ImR	The ImR (immediate register) holds the 16 bits of data pulled from instruction memory at PC+2. For I-type instructions, these 16 bits are the immediate value associated with the instruction.

- ALU

  The ALU will have two 16 bit input signals and one 3 bit control input signal. The output of the ALU is the result of the specified operation on the two 16 bit input signals. The list of operations that the ALU can perform are `add` , `sub`, `and`, `or`, and `slt` (this list is tentative). In the RTL, the ALU writes to ALUOut (for memory reference), Register File (for storing the result), and FCC (to increment and decrement the cache pointer for calls) .

- Comparator

  The comparator will be used in order to determine if two inputs are equal. It does this much quicker than the ALU would as it utilizes a series of 16 xor gates to try and 0 out a number and or's it with 0 at the end after all of the xors and inverts the result, yielding an isEqual result.

- Fcache

  The Fcache is a storage unit with 256 bit words that is used to automatically back up f registers. When another function is called, all f registers are immediately stored in this cache at the address of FCC. One 256 bit bus from registers 0 -256 and two control signals, a Fread and a Fwrite, are necessary. When Fread is high, then we will be taking in the f registers which are on the bus and storing them in the cache at FCC. When Fwrite is high, we are writing to all of the f registers with the data from the cache using the same 256 bit bus. These words will hold the 16 16-bit f registers specific to each allocated slot determined by the FCC.

- Dual-port Register File

  This component will be a file of 64 16 bit registers. It will take four inputs, two address inputs, and two data inputs. It will have two data outputs. The control signals will be RegReadA, RegReadB, RegWriteA, and RegWriteB. With a RegRead control signal high, the data at it's respective address input will be put on on it's respective output bus, and with the signal low, nothing happens. With a RegWrite control signal high, the data on it's respective input bus will be written to the address on it's respective address input.

- Dual-port Memory

  This component will be a memory unit with the ability to read and write two items at once. It will have four inputs, two address inputs, and two data inputs. It will have two data outputs. The control signals will be MemReadA, MemReadB, MemWriteA, and MemWriteB. With a MemRead control signal high, the data at it's respective address input will be put on on it's respective output bus, and with the signal low, nothing happens. With a MemWrite control signal high, the data on it's respective input bus will be written to the address on it's respective address input.

Testing our RTL

To verify the RTL for correctness, the RTL underwent.... 1. Peer review for any optimizations or lacking steps 2. A tracing of some simple algorithms to verify that the intended output was received.