# 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 in BAEJ architecture serve as registers which can be safely used during any function without backing up on a stack. In order to reduce the requirements imposed on the user, backing up and restoring the first 16 registers in our register file (.f0- .15) happens automatically to an internal memory unit, called the Fcache, upon function calls and returns. For the user to make a function call,

they simply need to move all of their values that they expect to be saved into the F registers. Upon return from the function call, the user's values will be safely returned to the F registers via the Fcache.

### 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 immedi- ate` | 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 im- mediate` | Changes pc to immediate | `pc=immediate` |
| `cal` | I Type | 0100 | `cal im- mediate` | Changes pc to immediate and sets a return address | `ra=pc+4pc=immedia` |
| `beq` | I Type | 0101 | `beq .rs .rd im- mediate` | Changes pc to immediate if rs and rd are equal | `if rs==rd pc=immedia` |

| Instruction | Type | OP | Usage | Description | Rtl |
|---|---|---|---|---|---|
| bne | I Type | 0110 | bne .rs .rd immediate | Changes pc to immediate if rs and rd aren't equal | if rs!=rd pc=immedia |
| 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

```c
// 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]<br>ImR = Mem[PC+2]<br>PC += 2 | | | | | | |
| PC += 2<br>A = Reg[IR[11:6]]<br>B = Reg[IR[5:0]] | | | | | PC = ImR | ra = PC + 2<br>PC = ImR |
| ALUout = A + ImR | | Reg[IR[5:0]] = ImR | if(A==B)<br>PC = ImR | ALUout = A << ImR | | Fcache[FCC] = Reg[15:0]<br>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]<br>PC += 2 | | | |
| A = Reg[IR[11:6]]<br>B = Reg[IR[5:0]] | | | PC = ra<br>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 | |

### Testing our RTL

- Peer review for any optimizations or lacking steps
- A tracing of some simple algorithms to verify that the intended output was received.

## Hardware Components

### Adder

| Items | Descriptions |
|---|---|
| **Inputs** | A[15:0], B[15:0] |
| **Outputs** | C[15:0] |
| **Control Signals** | None |
| **Functionality** | Outputs A+B onto C |
| **Hardware Implementation** | Series of half adders chain with carry outs to carry ins |
| **Unit Tests** | A loop in verilog that inputs all permutations of two 16 bit integers and verifies the output is the correct number you get when the inputs are added |

### Single bit Multiplexer

| Items | Descriptions |
|---|---|
| **Inputs** | A[15:0], B[15:0] |

| Items | Descriptions |
| --- | --- |
| **Outputs** | C[15:0] |
| **Control Signals** | Option |
| **Functionality** | Will constantly put the value of A or B specified by the Option control bit onto C |
| **Hardware Implementation** | Combinational logic to correctly choose the input to put on C |
| **Unit Tests** | A loop in verilog which puts every permutation of two 16 bit integers and a single control bit on A, B, and Option and tests that the output is whats expected |

**Two bit Multiplexer**

| Items | Descriptions |
| --- | --- |
| **Inputs** | A[15:0], B[15:0], C[15:0], D[15:0] |
| **Outputs** | C[15:0] |
| **Control Signals** | Option[1:0] |
| **Functionality** | Will constantly put the value of A or B specified by the Option control bit onto C |
| **Hardware Implementation** | Combinational logic to correctly choose the input to put on C |
| **Unit Tests** | A loop in verilog which puts every permutation of four 16 bit integers and a two bit control signal on A, B, C, D, and Option and tests that the output is whats expected |

**Register**

| Items | Descriptions |
| --- | --- |
| **Inputs** | A[15:0] |
| **Outputs** | B[15:0] |
| **Control Signals** | Write, Read |

| Items | Descriptions |
| --- | --- |
| **Functionality** | Stores value on A in latches on a Write signal, puts stored value on B on Read signal |
| **Hardware Implementation** | Series of individual latches with Write and Read signals distributed to them |
| **Unit Tests** | A loop in verilog that writes each value possible in a 16 bit integer and then reads them to ensure they are correct each iteration |

**ALU**

| Items | Descriptions | ALU op | Operation |
| --- | --- | --- | --- |
| **Inputs** | A[15:0], B[15:0] | 000 | AND |
| **Outputs** | A<B, C[15:0] | 001 | OR |
| **Control Signals** | Operation[2:0] | 010 | ADD |
| **Functionality** | Takes the mathematical operation specified by Operation and preforms in on operand A and B, puts result on A<B or C depending on operation | 011 | SUBTRACT |
| **Hardware Implementation** | A decoder used to interpret the control signal and combination logic unit for each of the operations availible | 100 | SHIFT |
| **Unit Tests** | A loop in verilog which inputs all permutations of two 16 bit integers and available operation codes and verifies with the output that the operation was preformed correctly on the inputs | 101 | SET LESS THAN |

**Comparator**

| Items | Descriptions |
| --- | --- |
| **Inputs** | A[15:0], B[15:0] |
| **Outputs** | C |
| **Control Signals** | Compare |

| Items | Descriptions |
| --- | --- |
| **Functionality** | Whenever the compare signal is high, outputs a 1 on C if A == B and a 0 otherwise. |
| **Hardware Implementation** | A tree of xor gates linking the bits of the inputs together down to a single bit which is inverted with an inverter gate |
| **Unit Tests** | A loop in verilog which inputs all permutations of two 16 bit integers and the Compare control signal and determines if the output is correct |

**Fcache**

| Items | Descriptions |
| --- | --- |
| **Inputs** | A[255:0], B[15:0] |
| **Outputs** | A[155:0] |
| **Control Signals** | Write, Read |
| **Functionality** | When the Write signal is high, takes the value on A and stored it in address B, when the Read signal is high, puts the value at B on A |
| **Hardware Implementation** | Static storage implemented using a register-file like structure. A decoder is used to decode the address to send the correct Write/Read signals to the correct address |
| **Unit Tests** | A loop in verilog which goes through each address and writes all possible 256 bit values while reading them each iteration to ensure they are correct. |

**Register File**

| Items | Descriptions |
| --- | --- |
| **Inputs** | A1[15:0], A2[15:0], W1[15:0], W2[15:0], F[255:0] |
| **Outputs** | R1[15:0], R2[15:0], F[255:0] |
| **Control Signals** | Write1, Write2, Read1, Read2, Backup, Restore |

| Items | Descriptions |
| --- | --- |
| **Functionality** | With a Write signal high, takes the respective value (W1 or W2) and stores it in the respective address (A1 or A2). With a Read signal high, takes the value at the respective address and puts it onto the respective output (R1 or R2). When Backup is high, puts the values in registers 0 to 15 on F, when Restore is high, stores the values on F into registers 0 to 15. |
| **Hardware Implementation** | Static storage implemented using a series of registers. A decoder is used to decode addresses and send the Write/Read signals to the correct register(s). |
| **Unit Tests** | A loop in verilog which goes through each address and writes all possible 16 bit values while reading them each iteration to ensure they are correct. |

**Memory Unit**

| Items | Descriptions |
| --- | --- |
| **Inputs** | A1[15:0], A2[15:0], W1[15:0], W2[15:0] |
| **Outputs** | R1[15:0], R2[15:0] |
| **Control Signals** | Write1, Write2, Read1, Read2 |
| **Functionality** | With a Write signal high, takes the respective value (W1 or W2) and stores it in the respective address (A1 or A2). With a Read signal high, takes the value at the respective address and puts it onto the respective output (R1 or R2) |
| **Hardware Implementation** | Memory unit using static storage and two decoders to implement the dual port functionality or sending the correct values and control signals to the correct registers. |
| **Unit Tests** | A loop in verilog which goes through each address and writes all possible 16 bit values while reading them each iteration to ensure they are correct. |

**Control Unit**

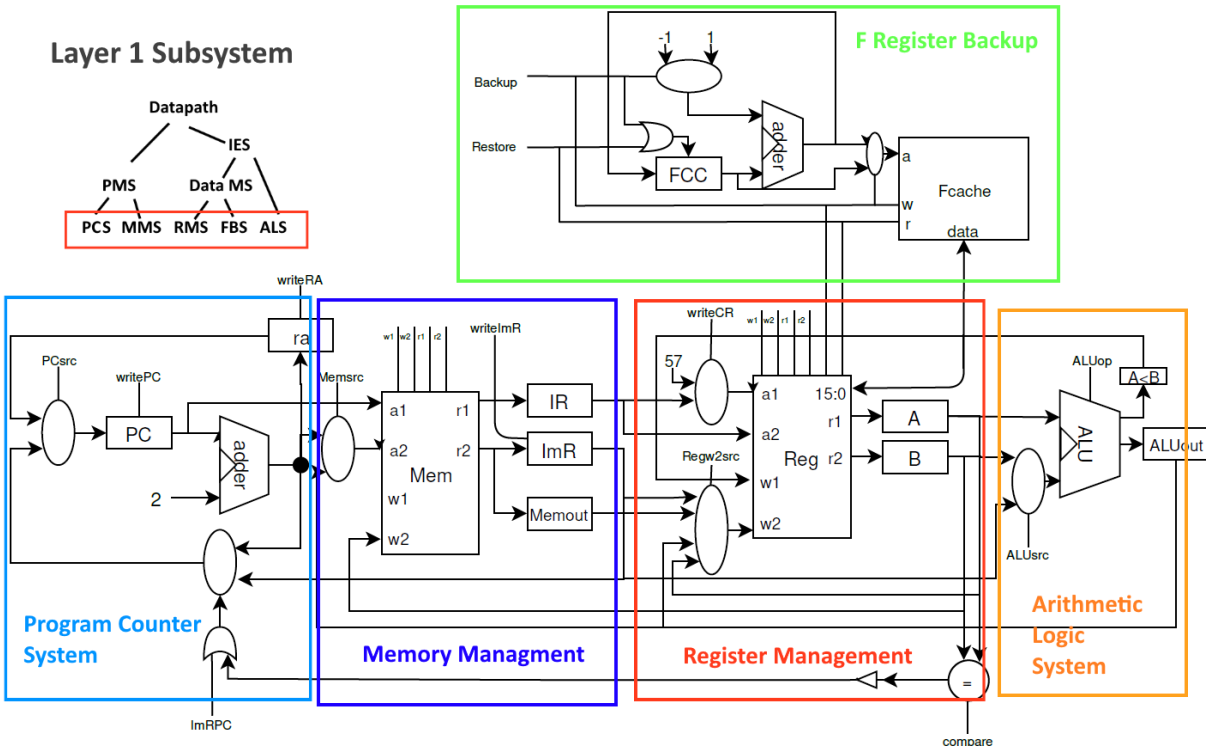
| Items | Descriptions |
| --- | --- |
| **Inputs** | A[3:0] |
| **Outputs** | B[23:0] |
| **Control Signals** | Reset |
| **Functionality** | Given an op-code (or address) the unit outputs a value on B corresponding to the control signals needed by the instruction |
| **Hardware Implementation** | Combinational logic to correctly choose the input to put on C |
| **Unit Tests** | A loop in verilog which puts every permutation of the op-codes and Reset control bit on A and Reset, then tests that the output control signals is whats expected |

## Integrating and Testing the Components

### Integration Plan

| Subsystem | Composition |
| --- | --- |
| **Program Counting System** | Register (x2), Single bit Multiplexer (x2), Adder, Or-gate |
| **Memory Management System** | Memory Unit, Single bit Multiplexer, Register (x3) |
| **Register Management System** | Register File, Single bit Multiplexer, Two bit Multiplexer, Register (x2) |
| **Fcache Backup System** | Fcache, Single bit Multiplexer (x2), Adder, Register, Or-gate |
| **Arithmetic and Logic System** | ALU, Single bit Multiplexer, Register (x2) |
| **Program Management System** | Program Counting System, Memory Management System |
| **Data Management System** | Register Management System, Fcache Backup System |
| **Instruction Execution System** | Data Management System, Arithmetic and Logic System |
| **Datapath** | Program Management System, Instruction Execution System |

### Datapath Block Diagram with Subsystems




### Test Plans

| Subsystem | Test Plan |
|---|---|
| **Program Counting System** (PCS) | Run the system through a few clock cycles to test that it correctly increments by two each time. Also ensure that we can write pc + 2 to ra. Once this is verified, inject addresses from a set of addresses, and from register ra, to test branching functionality. |
| **Memory Management System** (MMS) | Input values into a sequential block of memory then read from the same block, verifying that each read gives the output registers the correct values that were written. |
| **Register Management System** (RMS) | Input values into registers from all permutations of the input ports, then read from registers with known values verifying that each read gives the output registers the correct values. |

| Subsystem | Test Plan |
| --- | --- |
| **Fcache Backup System** (FBS) | Conduct multiple Backups of known values to a sequential block in the Fcache memory, then using multiple restores, read back the same block verifying the output is what was written. |
| **Arithmetic and Logic System** (ALS) | Conduct all possible ALU operations on a wide range of input values using all possible input methods (i.e. different ALUsrc signals to the multiplexer). Test each operation for correct output values. |
| **Program Management System** (PMS) | Hard-code values into a sequential block of memory then allow the program counter to increment through memory and verify that the correct values which were written to memory are written to the output registers. |
| **Data Management System** (DMS) | Repeatedly write values to registers 0 - 15 using many permutations of input methods. Each time all 16 registers are filled, send a backup control signal. Do this many times then conduct the same number of restores, ensuring values are correct along the way. |
| **Instruction Execution System** (IES) | Give this system the control signals needed for basic instructions which don't require memory such as arithmetic operations and moving values around in the register file. Include many different input values with each set of control signals and verify that the output is whats expected. |
| **Datapath** | Develop a set of test cases for each instruction and run them through the system and verify that the results are as expected. After these test have passed we can implement simple code blocks and algorithms to test more complex processes. |

### Control Signals

| Signal Name | Bits | Effect when deasserted (0) | Effect when asserted (1) |
|---|---|---|---|
| PCsrc | 1 | PC is set to default value (PC+2) or ImR | PC is set to the value of ra |
| writePC | 1 | Nothing | PC gets the value chosen by PCsrc mux |
| writeRa | 1 | Nothing | Ra gets the value of PC + 2 |
| ImPCsrc | 1 | ImPCsrc mux chooses PC+2 | ImPCsrc mux chooses immediate value (only when comparator is enabled and determines A=B) |
| MemSrc | 1 | Address 1 in Mem is pulled from PC + 2 | Address 1 in Mem is pulled from ALUout |
| MemW1 | 1 | Nothing | The value at port w1 is written to the address specified by a1 |
| MemW2 | 1 | Nothing | The value at port w2 is written to the address specified by a2 |
| MemR1 | 1 | Nothing | The value at the address specified by a1 is read to port r1 |
| MemR2 | 1 | Nothing | The value at the address specified by a2 is read to port r2 |
| writeCr | 1 | The reg number specified at reg file port a1 is IR[11:6] (default) | The reg number specified at reg file port a1 is 57 (for compiler register) |

| Signal Name | Bits | Effect when deasserted (0) | Effect when asserted (1) |
|---|---|---|---|
| RegSrc | 2 | 0 - Value at reg file port w2 comes from ImR; 1 - Value at port w2 comes from MemOut | 2 - Value at port w2 comes from ALUout; 3 - Value at port w2 comes from reg A |
| writeImR | 1 | Nothing | ImR gets the value read from memory at the address specified by a2 |
| backup | 1 | Nothing | Registers 15:0 (256 bits) from the reg file are written to the Fcache at the address specified by “a”; FCC is incrememented by 1 |
| restore | 1 | Nothing | The 256 bit value at the address specified by “a” in the Fcache is written to registers 15:0 in the reg file; FCC is decremented by 1 |
| RegW1 | 1 | Nothing | The value at port w1 is written to the reg address specified by a1 |
| RegW2 | 1 | Nothing | The value at port w2 is written to the reg address specified by a2 |
| RegR1 | 1 | Nothing | The value at the reg address specified by a1 is read to port r1 |

| Signal Name | Bits | Effect when deasserted (0) | Effect when asserted (1) |
|---|---|---|---|
| RegR2 | 1 | Nothing | The value at the reg address specified by a2 is read to port r2 |
| ALUsrc | 1 | 2nd ALU operand comes from ImR | 2nd ALU operand comes from reg B |
| ALUop | 3 | SEE ALU IN COMPONENTS | SEE ALU IN COMPONENTS |
| cmp | 1 | Nothing | The result of the comparison A=B is sent to the ImPCsrc mux |