By Thomas Bonatti

On behalf of: Josh, Ben Effron, Jeff, Thomas

**Addressable Registers**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Register** | **Address** | **Purpose** | **Register** | **Address** | **Purpose** |
| $M | 000 | Main accumulator. Used for arguments and returned values. | $t0 | 100 | General use. |
| $ra | 001 | Return address | $t1 | 101 | General use. Recommended to be used for flags |
| $sp | 010 | Stack pointer | $t2 | 110 | General use. Recommended for use as a second argument or return value. |
| $at | 011 | Pseudo instructions | $s1 | 111 | Safe use |

**Non-Addressable Registers:**

|  |  |
| --- | --- |
| **Register** | **Purpose** |
| PC | Points to current instruction. |

**I-Type instructions (Immediate):**

I-type instructions will use the main accumulator register, and a 12 bit immediate. Most logic and arithmetic commands will use this.

|  |  |
| --- | --- |
| Op Code [15:12] | Immediate [11:0] |

**R-type (move):**

Operations using registers

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Op Code [15:12] | rt (register) [11:9] | rs (register) [8:6] | Func code[5:2] | Unused [1:0] |

**IR-Type(Immediate and Register):**

Branches and load word store word

|  |  |  |
| --- | --- | --- |
| Op code [15:12] | Reg [11:9] | Immediate [8:0] |

**JR-Type [Jump Register]:**

Used for jump register command

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Op Code [15:12] | Reg [11:9] | Unused[8:6] | Func code[5:2] | Unused [4:0] |

Op codes:

|  |  |  |  |
| --- | --- | --- | --- |
| **Op Code** | **Operation** | **Op Code** | **Operation** |
| 0000 | Use function code\* | 1000 | li (load immediate) |
| 0001 | addi (add immediate) | 1001 | ori (or immediate) |
| 0010 | beq (brach equal) | 1010 | andi (and immediate) |
| 0011 | bne (branch not equal) | 1011 | nori (nor immediate) |
| 0100 | J (jump) | 1100 | sll (shift left logical) |
| 0101 | lw (load word) | 1101 | srl (shift right logical) |
| 0110 | sw (store word) | 1110 | sra (shift right arithmetic) |
| 0111 | lui (load upper immediate) | 1111 | slti (set less than) |

**Function Codes:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Func Code** | **Operation** | **Func Code** | **Operation** |
| 0000 | add | 1000 |  |
| 0001 | and (logic) | 1001 | slt (set less than) |
| 0010 | or (logic) | 1010 | sub (subtract) |
| 0011 | xor (logic) | 1011 | brqz (branch if zero) |
| 0100 | nor (logic) | 1100 | bnez (branch not zero) |
| 0101 |  | 1101 |  |
| 0110 | copy | 1110 |  |
| 0111 | jr | 1111 |  |

**RTL table**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **R-type (ALU)** | **I-type (ALU)** |  |
| **Inst. Fetch** | IR = mem[PC]  PC = PC + 1 | | |
| **Inst. Decode** | AiA = Reg[IR [7:5] ]  BiB = Reg[IR [4:2] ] | AiA = Reg [0]  BiB = SE [IR [11:0] ] |  |
| **Execution** | ALUout = AiA op BiB | |  |
| **Mem / Reg access 1** | Reg [0] = ALUout | |  |

**Instruction Descriptions**

**Check function code** (op = 0000)

If the op code is 0000, the operation to be executed will be determiner by a 4 bit function code [11:8]

**Addi** **– add immediate** (op 0001, func = xxxx)

signature: addi <12 bit imm.>

I-type

Addi is an I-type operation that adds the given immediate to the accumulator register.

**Beq – Branch equal**  (op = 0010, func = xxxx)

signature: beq $r, <9 bit imm.>

IR-type

Beq branches from PC + 1 if the given register $r and the main accumulator are equal. Can branch +/- 256 lines up or down.

**Bne** - **brach not equal**(op = 0011, func = xxxx)   
signature: beq $r, <9 bit imm.>

IR-type

Bne branches if the given register $r and the main accumulator are not equal. Can branch ± 256 lines from PC+1.

**J – jump** (op = 0100, func = xxxx)

Signature: j <12 bit imm.>

I-type

The 4 most significant digits in the PC will be concatenated with a 12 bit immediate and used as the new PC.

**lw – load word** (op = 0101, func = xxxx)

signature: lw $r[ <9 bit imm.> ]

IR-type

Loads a 16 bit chunk of data from memory at the given address incremented by the immediate.

**sw** **– store word** (op = 0110, func = xxxx)

signature: sw $r [ <9 bit imm.> ]

IR-type

Stores the value in the main register in memory at the given address, incremented by the immediate.

**Li – Load Immediate** (op = 1000, func = xxxx)

Signature: li <12 bit imm.>

I-type

Li stores the given immediate value in the main register.

**ori – or immediate** (op = 1001, func = xxxx)

signature: ori <12 bit imm.>

I-type

ori performs a logical or operation on the value of the main register and the immediate, then stores the value in the main register.

**andi – and immediate** (op = 1010, func = xxxx)

signature: andi <12 bit imm.>

I-type

andi performs a logical and operation on the value of the main register and the immediate, then stores the value in the main register.

**nori – nor immediate** (op = 1011, func = xxxx)

signature: nori <12 bit imm.>

I-type

nori performs a logical nor operation on the value of the main register and the immediate, then stores the value in the main register.

**sll – shift left logical** (op = 1100, func = xxxx)

IR-type

signature: sll $r, <9 bit imm.>

**srl – shift right logical** (op = 1101, func = xxxx)

IR-type

signature: srl $r, <9 bit imm.>

**sra – shift right arithmetic** (op = 1110, func = xxxx)

IR-type

signature: sra $r, <9 bit imm.>

**slti – set less than** (op = 1111, func = xxxx)

signature: slt $r, <9 bit imm.>

I-type

Sets the given register r to 1 if the main register $m is less than the 9 bit immediate. Else, r is set to 0.

**add – add** (op = 0000, func = 0000)

signature: add $r1, $r2

R-type

Add adds $r1 and $r2 together and stores in the main register. To accumulate, use add $m, $r1.

**and – and** (op = 0000, func = 0001)

signature: and $r1, $r2

R-type

Preforms a logical and operation on two registers, then stores the result in the main register.

**or – or** (op = 0000, func = 0010)

signature: or $r1, $r2

R-type

Preforms a logical or operation on two registers, then stores the result in the main register.

**xor – xor** (op = 0000, func = 0011)

signature: xor $r1, $r2

R-type

Preforms a logical xor operation on two registers, then stores the result in the main register.

**nor – nor** (op = 0000, func = 0100)

signature: nor $r1, $r2

R-type

Preforms a logical nor operation on two registers, then stores the result in the main register.

**copy – copy** (op = 0000, func = 0110)

signature: copy $r1, $r2

R-type

Writes the value of $r1 into register $r2.

**jr – jump register** (op = 0000, func = 0111)

signature: jr $r

JR-type

jumps to the address in the given register.

**slt – set less than register** (op = 0000, func = 1001)

signature: sltr $r1, $r2

R-type

If the main register is less than register $r1, a flag is set to 1 in register $r2. If the main register is equal to or greater than register $r1, a flag is set to 0 in register $r2.

**sub – subtract** (op = 0000, func = 1010)

signature: sub $r1, $r2

R-type

Does operation $m = r1 – r2.

**beqz – branch if zero** (op = 0000, func = 1011)

Signature: beqz <12 bit imm.>

I-type

If the main register equals zero, branches. This branch command can branch ±2048 instructions from PC+1

**bnez– branch if zero** (op = 0000, func = 1100)

Signature: bnez <12 bit imm.>

I-type

If the main register does not zero, branches. This branch command can branch ±2048 instructions from PC+1

**Pseudo Instructions**

**Jal – Jump and Link**

Jump and link jumps to an immediate after backing up the return address and setting a new return address in $ra.

**Instructions**:

swap $m, $ra # saves $m # 0000 0101 000 001 xx

sw $sp[0] # puts the return address on the stack. # 0110 010 000000000

swap $m, $sp # 0000 0101 000 010 xx

addi 1 # increments the stack pointer # 0001 000000000001

swap $m, $sp # gets $ra back. # 0000 0101 000 010 xx

Swap $m, $ra # gets original M back. # 0000 0101 000 001 xx

j <12 bit imm.> # jump to the given immediate # 0100 <12 bit imm.>

**Swap – swap registers**

Swaps the values between registers

**Instructions**:

Copy $m, $at

Copy $m, $R

Copy $at, $m

**Lui – Load upper immediate**

**Move – moves a value from one register to another AND zeros out the source register**

**Datapath Components:**

**PC:** PC is the pointer counter register. It outputs the current instruction address, then stores the next instruction address. It takes in and outputs bits [15:0]. PC also has a PC write signal which prevents the PC to be incremented during cycles when it should not be incremented.

**Adder1**: This adder takes in the current instruction address and adds 1 to it. Both input and output are 16 bits.

**Adder2:** This adder is used for branches and adds the sign extended immediate value from the instruction with the PC. However, this happens AFTER PC has been incremented for this cycle, so the jump will really be from PC of current instruction + branch value + 1

**Instruction Memory:** Instruction memory is a block of memory which contains the instructions in the program. It takes in the current instruction address and outputs the instruction. Takes in and outputs bits [15:0]

**IR:** IR is the instruction register. It takes the instruction output from the Instruction Memory, holds it through the rest of the cycle, and then on the next cycle outputs the instruction that is was fed in. It takes in and outputs bits [15:0]

**Register File:** The bits from the instruction are split up and sent to the appropriate areas of the Register File. The Register File takes in the addresses of the relevant registers from the instruction and outputs the contents of those registers. It also takes in either the resulting value from an operation or an immediate value. The result of all computations are stored in main, so there is no destination register input. The register file outputs the contents stored in the specified registers. The register file has one control signals: reg. Regw is the signal for register writing. If it is turned on the value output by the main ALU is stored in the main register. If it is off writing to the main register is disabled. The ra1 takes in bits [7:5] which chooses the register in both R and IR type instructions. Ra2 takes in bits [4:2], which are chooses the second register for R type instructions. The register file outputs two sets of 16 bits, the contents of the registers.

**ALUinA, ALUinB:** These are both registers that take the output from the register file to store the values between cycles and then output those values to the ALU. Takes in and outputs 16 bits.

**ALU Main:** ALU Main is the main ALU in the processor. The values stored in ALUinA, ALUinB, the sign extended immediate value are inputs for the ALU, and are chosen depending on the type of instruction. The ALU outputs the result of the computation. The ALU takes in ALU control, which tells the ALU which operation to perform. ALU main takes in two sets of 16 bits and outputs a 16 bit answer. It also outputs a 1 bit, isZero which is used for branches.

**ALUOut:** ALUOut is the register that stores the resulting calculations from the ALU. ALUOut stores these results between cycles and outputs the results at the beginning of the next cycle. It takes in the 16 bit results and outputs a 16 bit result.

**Data Memory:** Data memory is the main memory in the computer. Data memory takes in the result from the ALU either retrieves the value stored at that location in memory or stores the value in ALUinA (the value of the main register) at the location in memory in the specified register, and outputs that value. Data memory has a mem write control signal, which allows writing to memory on the correct instructions and turning off writing to memory on instructions that are not supposed to write to memory. Data memory takes in a 16 bit number and outputs the 16 bit result from the read.

**MemOut:** A register that holds the 16 bit output of reads from memory through cycles, and outputs the value stored in it.

**Various Components:** A variety of muxs and sign extends are in use.

Register Transfer Language

\*Ai = ALUin

\*IR-Type is split up between lw/sw, jump, branch, and copy

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **R-Type** | | **I-Type** | **Load Imm.** | **Jump** | **Branch** | **Copy** | **Load Word** | **Store Word** |
| **Inst. Fetch** | IR = Mem[PC]  PC=PC+1 | | | | | | | | |
| **Inst. Decode**  **Reg. Fetch** | AiA = Reg[IR[7:5]]  AiB = Reg[IR[4:2]] | | AiA = Reg[$m]  AiB = SE[IR[11:0]] | Reg[0] = SEim [15:0] | PC = PC [15:12] concat. IR [11:0] | AiA = Reg[0]  AiB = Reg[IR[11:9]] | copyReg = Reg[IR[7:5]] | AiA = Reg[IR[7:5]]  AiB = SE[IR[9:0]] | AiA = Reg[IR[7:5]]  AiB = SE[IR[9:0]] |
| **Execution** | ALUout = AiA op AiB | | | Li DONE | Jump Done | If (AiA == AiB)  Then: PC = PC + IR[8:0]  Else: PC = PC | Reg[IR[4:2]] = copyReg | ALUout = AiA + AiB | ALUout = AiA + AiB |
| **Reg/mem Access** | Reg[0] = ALUout | | |  |  | Branch DONE | Copy DONE | MemOut = Mem[ALUout] | Mem[ALUout] = reg[0] |
| **Reg/mem 2** | R-type DONE | I-type DONE | |  |  |  |  | Reg[0] = memOut | sw DONE |
| **Lw Done** |  | |  |  |  |  |  | lw DONE |  |

RTL test plan

This documentation walks through brief examples of how each command in the RTL will work.

**R-Type (ALU Commands)**:

**Initialize:** $m = 10 (0b1010), $t0 = 5 (0b101), PC = 0

**Instructions**: Add $m, $t0 (0000 0000 000 100 XX)

And $m, $t0 (0000 0001 000 100 XX)

Add $m, $t0

**Fetch:** IR = 0000 0000 000 100 XX

PC = 0 + 1

**Decode:** AiA = 10

AiB = 5

**Execute:** ALUout = 10 + 5 = 15

**Mem/Reg:** Reg[0] = 15

And $m, $t0

**Fetch:** IR = 0000 0001 000 100 XX

PC = 1 + 1

**Decode**: AiA = 15 (0b1111)

AiB = 5 (0b0101)

**Execute**: ALUout = 1111 && 0101 = 0101

**Mem/Reg**: Reg[0] = 5

**I-type (ALU commands)**:

**Initialize**: $m = 0, PC = 10

**Instructions**: Addi 15

Ori 32

Addi 15

**Fetch**: IR = 0001 000000001111

PC = 10 + 1 = 11

**Decode**: AiA = 0b0000000000000000

AiB = 0b0000000000001111

**Execute**: ALUout = 15 + 0 = 15

**Mem/Reg**: reg [0] = 15

Ori 32

**Fetch**: IR = 1001 000000100000

PC = 11 + 1 = 12

**Decode**: AiA = 0b0000000000001111

AiB = 0b0000000000100000

**Execute:** ALUout = 15 || 32 = 0000000000101111 = 47

**Mem/Reg:** reg [0] = 47

**Load Immediate**

**Initialize**: $m = 0, PC = 10

**Instructions**: Li 10

Li12

**Fetch**: IR = 1000 000000001010

PC = 10 + 1 = 11

**Decode**: Reg[0] = 10

**Fetch**: IR = 1000 000000001100

PC = 11 + 1 = 12

**Decode**: Reg[0] = 12

**Jump**

**Initialize**: PC = 0x014a

**Instructions**: j 10

J 100

j 10

**Fetch**: IR = 0100 000000001010

PC = 0x014a + 1 = 0x014b

**Decode**: PC = 0x0 concatenate 0x00a = 0x000a

j 100

**Fetch**: IR = 1000 000000001100

PC = 0x000a + 1 = 0x000b

**Decode**: PC = 0x0 concatenate 0x064 = 0x0064

**Branch**

**Initialize**: $m = 0, $t0 = 0, $t1 = 10 PC = 10

**Instructions**: beq $t0, 15

beq $t1, 32

beq $t0, 15

**Fetch**: IR = 0010 000000001111

PC = 10 + 1 = 11

**Decode**: AiA = 0b0000000000000000

AiB = 0b0000000000000000

**Execute**: ALUout = 0 - 0 = 0

PC = 11 + 15 = 26

beq $t1, 32

**Fetch**: IR = 0010 000000100000

PC = 26 + 1 = 27

**Decode**: AiA = 0b0000000000000000

AiB = 0b0000000000001010

**Execute:** ALUout = 0 – 10 = -10

PC = 27