CPU Design Project Part 1 - ISA Design

Joe Driscoll

February 1, 2019

Contents

1 Registers

The current CPU design has a total of sixteen registers, thirteen of which are directly programmable and three of which are used by CPU. Sixteen registers were used to make the bit-alignment in the machine instructions more convenient, as in instructions that use three register operands like "add", the operands perfectly fill up the sixteen-bit word. The names of the registers, as well as their designated uses are shown in the table below.

Table 1: CPU registers.

Register(s)	Number	Purpose
zero	0	zero constants/offsets (frequently used)
s0 - s5	1-6	general-purpose and parameter/return passing
r0 - r7	7-14	general-purpose
lsr	15	used to return from subroutines

2 Instruction formats

There are four instruction formats that the CPU uses, namely R-type, J-type, I_1-type, and I_2-type. These are similar to the MIPS instruction format specifications (R, I, and J types), except the number subscript on the I-type instruction indicates how many registers are expected inside the machine instruction. Also, the most significant bit in the immediate operands of all instruction formats are aligned to simplify instruction decoding.[1]

2.1 R-type

R-type instructions are instructions that have a register number for all operands. Because instructions are sixteen-bits and opcodes are four-bits, these instructions contain only the opcode and three register operands. The format is shown below.

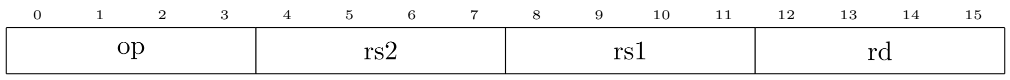

Figure 1. Instruction format for R-type instructions.

2.2 J-type

J-type instructions are reserved specifically for branching instructions, and there are no register operands in this format. This format contains the four bit opcode, a two bit conditional, and a ten bit address, meaning the CPU can only branch to labels. It was difficult to imagine a scenario where branching from or relative to another register would be useful. If it turns out this is a common use case, then modifications will be made to the J-type instructions. The exact format of the instruction is shown below.

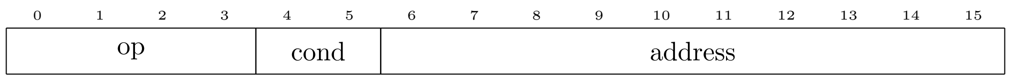

Figure 2. Instruction format for J-type instructions.

[1]Due to Matthew Cather - this optimization trick came up in casual conversation about instruction decoding.

2.3 I_1-type

I_1-type instructions contain the opcode, one register operand, and an eight-bit constant operand. These are really just used for loading in larger constants to registers, allowing for sixteen-bit constants to be loaded into registers in two instructions. These are not really essential, but they provide convenience in these relatively niche cases.

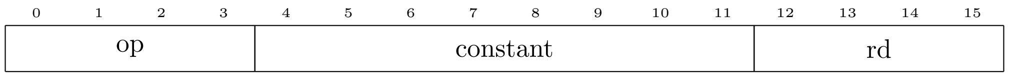

Figure 3. Instruction format for I_1-type instructions.

2.4 I_2-type

I_2-type instructions contain the opcode, two register operands, and a four-bit constant operand. These allow things like adding and subtracting constants, as well as bit-shifting a constant number of times. Again, these are not exactly required, but they make relatively common operations (like doing arithmetic with constants) simple.

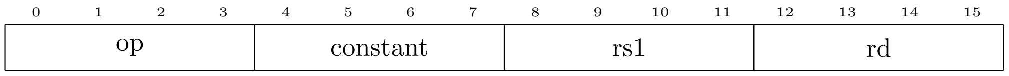

Figure 4. Instruction format for I_2-type instructions.

3 Instruction set

Table 2 contains all sixteen of the ISA's instructions, as well as the assembly and machine language opcodes and instruction format for each. For each of these instructions, the registers are encoded by their register number (four bits in the instruction), and constants are encoded by their value in binary. The only thing that varies from instruction to instruction is the opcode and the instruction format, and these are indicated in the table. There is only one instruction that is an exception to the instruction format, and that is the cmp instruction. The cmp instruction does not use the first register operand (destination register) as it uses the lower three bits of the program counter to store information about the results of the comparison. Refer to section two for the exact bitfield location for each instruction format. Table 3 contains the motivations for the inclusion of each instruction. Each instruction is included in this section with a brief explanation of what exactly the instruction does. The description of each instruction includes the instruction format for quick reference.

One requirement of this instruction set was the inclusion of a halt instruction. While no instruction is designed specifically for halt, halt can be considered a pseudo-instruction that simply translates to "b label_above_this_instruction". In other words, wherever a halt instruction in assembly code can be translated to a label followed by a branch to that label. No special accommodations must be made for the branch instruction, but the symbol table would need to be updated to keep track and determine the address of the label from the translation.

Table 2: Assembly format, opcode, and format type for each instruction.

Instruction	Assembly	Opcode	Instruction format
add	add \$rA[2], \$rB, \$rC	0000	R-type
sub	sub \$rA, \$rB, \$rC	0001	R-type
str	str \$rA, \$rB, \$rC	0010	R-type
ldr	ldr \$rA, \$rB, \$rC	0011	R-type
and	and \$rA, \$rB, \$rC	0100	R-type
or	or \$rA, \$rB, \$rC	0101	R-type
not	not \$rA, \$rB	0110	R-type
cmp	cmp \$rA, \$rB[3]	0111	R-type
br[4]	br \$rA	1000	R-type
b	b(,eq,lt,gt)[5] label[6]	1001	J-type
bl	bl(,eq,lt,gt) label	1010	J-type
loadil	loadil \$rA, kk[7]	1011	I_1-type
loadiu	loadiu \$rA, kk	1100	I_1-type
addi	addi \$rA, \$rB, k[8]	1101	I_2-type
lsr	lsr \$rA, \$rB, k	1110	I_2-type
lsl	lsl \$rA, \$rB, k	1111	I_2-type

[2]An $r(letter) denotes a general-purpose register.

[3]As mentioned above, cmp is the only exception to the R-type format, as it does not use the destination register operand. It only modifies three of the lower significant bits of the program counter.

[4]In this case, rA is the only register operand used - rA contains the register whose contents point to the address to branch to (e.g. lsr).

[5]The different types of conditional branches are noted in parentheses. Branches can be done unconditionally, on equality, less than, or greater than.

[6]A label denotes a label or symbol in the symbol table (absolute address).

[7]The letters kk denote an eight-bit constant.

[8]The letter k denotes a four-bit constant.

Table 3: Motivations behind the inclusion of each instruction in the ISA.

Instruction	Motivation
add	Adding is very common, especially in adding two registers.
sub	Same reason as add.
str	Storing the value of a register to memory is needed for working with data in memory.
ldr	Loading the value of a memory location to a register is fundamental to working with data in memory.
and	Logical and's are one of the most common bit-wise operations (bit masking).
or	Logical or's are another of the most common bit-wise operations (bit setting).
not	Logical "not" combined with or's and and's allows for any logical expression to be composed.
cmp	Comparing the value of two registers is necessary for making useful control structures for branching, loops, etc.
br	Branching to registers is useful when returning from subroutine calls, and perhaps in other strange edge cases that a compiler could devise to optimize something.
b	Branching is a fundamental operation that allows control structures to exist in a convenient way.
bl	Being able to branch and return to that branch point allows for effective inclusion of subroutines in assembly programming.
loadil	Combining with load immediate upper (loadiu) allows sixteen-bit constants to be loaded into registers.
loadiu	Combining with load immediate lower (loadil) allows sixteen-bit constants to be loaded into registers.
addi	Adding a constant value to a register is another quite common operation (like incrementing a register by 1 in control structures).
lsr	Logical shifts by a constant amount are common. The four-bit constant operand allows shifting over the full range of the sixteen-bit register.
lsl	Same reason as lsr.

3.1 add

The add instruction adds the contents from rs1 and rs2 and stores the result in rd. Each of these registers are identified by their register number in the machine code.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0000				rs2				rs1				rd			

Figure 5. Instruction format for add.

3.2 sub

The sub instruction subtracts the contents of register rs2 from rs1 and stores the result in rd.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0001				rs2				rs1				rd			

Figure 6. Instruction format for sub.

3.3 str

The str instruction stores the contents of rd into the memory location specified by rs1, offset by rs2. If no offset is needed, then rs2 can be the zero register for a zero offset. The offset of rs2 is typically used for things like array parsing (e.g. expressions like A[i] = B[i++]).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0010				rs2				rs1				rd			

Figure 7. Instruction format for str.

3.4 ldr

The ldr instruction loads the memory contents at memory location rs1 offset by rs2 into rd. This works the same as str, except it is fetching from memory rather than storing. Again, the $ zero register can be used for no offset memory fetches.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0011				rs2				rs1				rd			

Figure 8. Instruction format for ldr.

3.5 and

The and instruction bit-wise AND's the contents of registers rs1 and rs2 and stores the result in rd. The bit-wise and is commutative, so order of the source operands does not matter.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0100				rs2				rs1				rd			

Figure 9. Instruction format for and.

3.6 or

The or operation performs a bit-wise OR on rs1 and rs2 and stores the results in rd. Again, a logical OR is commutative so the order of source operands does not matter.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0101				rs2				rs1				rd			

Figure 10. Instruction format for or.

3.7 not

The not operation negates the contents of rs1 and stores the result in rd.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0110				unused				rs1				rd			

Figure 11. Instruction format for not.

3.8 cmp

The compare instruction subtracts rs1 from rs2, storing information about the results in the lower three bits of the program counter register. PC[0] is 1 if and only if rs1 == rs2 and 0 otherwise, PC[1] is one if and only if rs1 > rs2 and 0 otherwise, and PC[2] is 1 if and only if rs1 < rs2 and 0 otherwise. This was done to utilize some of the otherwise wasted space in the program counter, as six of the sixteen bits of the register would be unused since the memory is only ten-bit. This also allows for relatively simple and expressive encoding of conditional branches, as well as the ability to preserve comparison results on function call and returns. Because bl stores the value of PC to later jump back to, the comparison information is not lost (assuming nested function calls are not performed, which would then require a stack).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0111				rs2				rs1				unused			

Figure 12. Instruction format for cmp.

3.9 br(,eq,lt,gt)

The br instruction branches to the address contained in the contents of register rd. Like the other branch instructions, a condition can be selected for a condition branch.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1000				cond		unused		rs1				unused			

Figure 13. Instruction format for br.

3.10 b(,eq,lt,gt)

The branch instruction simply branches to the ten-bit label provided in the instruction. There are multiple forms of the instruction, namely b, beq, blt, and bgt. These all use the opcode, but they have different values in the two-bit cond field. The cond codes for the conditions listed in the section heading are 00, 01, 10, and 11 respectively. These different conditions are checked by looking at the lower three bits of the program counter, as discussed in the section discussing the cmp instruction.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1001				cond		address									

Figure 14. Instruction format for b.

3.11 bl(,eq,lt,gt)

This instruction does the same thing as the b instruction, except it stores the program counter incremented one address (+0x40) into the lsr. Using this allows the program to branch back to where the branch was originally called from, which is very useful for function/subroutine calls.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1010				cond		address									

Figure 15. Instruction format for bl.

3.12 loadil

The loadil instruction loads an eight-bit constant kk into the lower-eight bits of the register specified by rd. This is used for loading in large constants that cannot fit in the four bits of the constant field in, say, addi.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1011				constant								rd			

Figure 16. Instruction format for loadil.

3.13 loadiu

The loadiu instruction does the same thing as loadil instruction, except the eight-bit constant is loaded into the upper-eight bits of the register instead of the lower-eight bits.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1100				constant								rd			

Figure 17. Instruction format for loadiu.

3.14 addi

The addi instruction adds the four-bit constant k to rs1 and stores the result in rd. This is mostly for things like incrementing a variable and loading small, four-bit constants by adding to the $zero register.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1101				constant				rs1				rd			

Figure 18. Instruction format for addi.

3.15 lsr

The lsr instruction shifts the contents of rs1 to the right by the constant number of bits specified in k and stores the result in rd.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1110				constant				rs1				rd			

Figure 19. Instruction format for lsr.

3.16 lsl

The lsl instruction does the same thing as lsr, except the shift is to the left instead of the right.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1111				constant				rs1				rd			

Figure 20. Instruction format for lsl.

4 C examples

This section contains some code examples that show how some of the fundamental C-language constructs would be compiled into this instruction set. The code examples are split up into variable assignments and expressions, control structures, and function calls. To examine how these instructions would be assembled into machine language, refer to the previous sections on instruction formats and the various opcodes available.

4.1 Assignments and expressions

The below two pieces of code show how simple arithmetic and logical expressions would be compiled into this assembly language.

```
int a = 0;
int b = 0;
int c;

a = a + 1;
b = b - 1;
c = a + a + a - b - b;
```

```
addi $r0, $zero, 0x0    ; Initialize a in $r0.
addi $r1, $zero, 0x0    ; Initialize b in $r1.
addi $r2, $zero, 0x0    ; Initialize c in $r2.

addi $r0, $r0, 0x1      ; a = a + 1
addi $r1, $r1, -0x1     ; b = b - 1
add  $r3, $r0, $r0      ; c = a + a
sub  $r4, $r0, $r1      ; $r4 = a - b
sub  $r4, $r4, $r1      ; $r4 = a - b - b
add  $r3, $r3, $r4      ; c = a + a + a - b - b

end:
    b end               ; End program.
```

Figure 21. Compilation of addition and subtraction.

```
int ones, mask, masked, new_ones;
ones = 0xFF;
mask = 0xAA;

/* Mask alternating bits. */
masked = ones & mask;

/* Set previously masked bits. */
new_ones = masked | (~mask);
```

```
addi $r0, $zero, 0x0    ; Initialize ones in $r0.
addi $r1, $zero, 0x0    ; Initialize mask in $r1.
addi $r2, $zero, 0x0    ; Initialize masked in $r2.
addi $r3, $zero, 0x0    ; Initialize new_ones in $r3.
; Set ones to 0xFF;
loadiu $r0, 0xF         ; Set upper half of $r0 to 0xF
loadil $r0, 0xF         ; Set lower half of $r0 to 0xF
; Set mask to 0xAA;
loadiu $r1, 0xA         ; Set upper half of $r1 to 0xA
loadil $r1, 0xA         ; Set lower half of $r1 to 0xA
and $r2, $r0, $r1       ; Apply mask to ones.
not $r4, $r1            ; Negate mask.
or  $r3, $r2, $r4       ; Set masked bits.
```

Figure 22. Compilation of logical expressions.

4.2 Control structures

These code segments show how some of the common control structures like if-else, while, and for loops are compiled in this assembly language. The first example, which shows if-else structures, also demonstrates all six of the required logical expressions (==, !=, >, <, >=, <=). As a result, the while and for examples only show a subset of these for brevity, considering that the first example shows the full range of logical comparison capabilities. None of the code examples do anything exceptional or interesting - they simply demonstrate the ability to use control structures.

4.2.1 If-else structure

This large code sample shows how each of the different logical comparisons would be compiled. There are four main sections to the code (separated paragraphs). The first three blocks correspond to the first three if-else statements, while the last is simply the instruction that halts the program (could be interpreted from a "halt" pseudo-instruction).

```c
int a;

if (a > 1)
    a++;
else if (a < 1)
    a--;

if (a >= 2)
    a--;
else if (a <= 2)
    a++;

if (a == 3)
    a = a + 2;
else if (a != 3)
    a = a - 2;
```

```
    addi $r0, $zero, 0x0    ; Initialize a to 0.
    addi $r1, $zero, 0x1    ; Load compare value.
    cmp  $r0, $r1           ; Compare a to 1.
    blt  label_1            ; Check if less than.
    beq  label_2            ; Check if equal to.
    addi $r0, $r0, 0x1      ; If gt, add 1.
    b label_2               ; Go to next if.
label_1:
    addi $r0, $r0, -0x1     ; If lt, subtract 1.

label_2:
    addi $r1, $zero, 0x2    ; Load next compare value.
    cmp  $r0, $r1           ; Compare a to 2.
    blt label_3             ; Check less than.
    addi $r0, $r0, -0x1      ; If geq, subtract 1.
    b label_4               ; Go to next if.
label_3:
    addi $r0, $r0, 0x1      ; If leq, add 1.

label_4:
    addi $r1, $zero, 0x3    ; Load next compare value.
    cmp  $r0, $r1           ; Compare a to 3.
    beq  label_5            ; Check equality.
    addi $r0, $r0, -0x2     ; If neq, subtract 2.
    b end                   ; Go to end.
label_5:
    addi $r0, $r0, 0x2      ; If eq, add 2.

end:
    b end
```

Figure 23. Compilation of an if-else structure that also shows all logical comparison types.

4.2.2 While structure

This code shows a basic example of using a while loop, demonstrating that the assembly language can in fact compile a C-language while statement. In this case, it is assumed that the compiler has determined a location in memory for the array a, and the pointer to this array is assumed to be in $r0 at the beginning of the assembly program.

```
                                   addi   $r1, $zero, 0x0    ; Set i to 0 in $r1.
int i = 15;                        loadil $r1, 0xF           ; Set i to 15 in $r1.
int a[15];                     while_loop_1:
                                   cmp    $r1, $zero         ; Compare i to 0.
while (i >= 0)                     blt    end                ; Exit if not geq.
{                                  str    $r1, $r0, $r1      ; Store i in ith entry of a.
    a[i] = i;                      addi   $r1, $r1, -0x1     ; Decrement i.
    i--;                           b      while_loop_1       ; Repeat loop.
}
                               end:
                                   b    end                ; End program
```

Figure 24. Compilation of a while control structure.

4.2.3 For structure

This example implements the same program shown in the while structure section, except it uses a for loop to accomplish the task, and it stores the values in the opposite order. The same assumption is made about the address of array a being placed in $r0 at the time of execution.

```
                                   addi $r1, $zero, 0x0    ; Initialize i to 0.
                                   addi $r2, $zero, 0x0    ; Initialize comparison value.
int i;                             loadil $r2, 0xF         ; Set comparison value to 15.
int a[15];                     for_loop_1:
                                   cmp $r2, $r1            ; Compare 15 to i.
for (i = 0; i <= 15 0; i++)        bgt end                 ; If i < 15, exit loop.
{                                  str $r1, $r0, $r1       ; Store i in i-th entry of a.
    a[i] = i;                      addi $r1, $r1, 0x1      ; Increment i.
}                                  b for_loop_1            ; Restart loop.

                               end:
                                   b end                   ; End program.
```

Figure 25. Compilation of a for control structure.

4.3 Function calls

This code shows a simple example showing how C-language function calls are compiled in this assembly language. This demonstrates the purpose of the s registers, which can be used as general-purpose registers, but they are also used for parameter and return value passing. More specifically, functions expect s0-s3 to contain function parameters while the caller expects the return value to be in s4. This is the ISA ABI (Application Binary Interface) for function calls. This example does not show an argument passed by reference, but this could be done in the same way as shown below, except the address of a variable is placed in one of the ABI parameter registers ($s0-$s3).

```
/* Doubles each argument and
   returns their sum. */
/* Assumes no overflow error
   will occur. */
int double_and_add(int a, int b)
{
    int answer = a + a + b + b;
    return answer;
}

void main()
{
    int a, b, c;
    a = 5;
    b = -7;
    c = double_and_add(a, b);
}
```

```
main:
    addi $r0, $zero, 0x5     ; Store a in $r0.
    addi $r1, $zero, -0x7    ; Store b in $r1.

    ; Function call preamble.
    addi $s0, $r0, 0x0       ; Parameter 1.
    addi $s1, $r1, 0x0       ; Parameter 2.

    bl double_and_add        ; Function call.

    ; Move result to $r2 for hypothetical later use.
    addi $r2, $s4, 0x0

end:
    b end                    ; End program.

double_and_add:
    add $r2, $s0, $s0        ; $r2 = a + a
    add $r3, $s1, $s1        ; $r3 = b + b
    add $s4, $r2, $r3        ; $s4 = 2a + 2b
    br  $lsr                 ; Return from call.
```

Figure 26. Compilation of a function with parameters and return value.