

Computer Architecture

Lecture No. 8

Reading Material

Handouts

Slides

Summary

- 1) Introduction to the ISA of the FALCON-A
- 2) Examples for the FALCON-A

Introduction to the ISA of the FALCON-A

We take a look at the notation that we are going to employ when studying the FALCON-A. We will refer to the contents of a register by enclosing in square brackets the name of the register, for instance, R [3] refers to the contents of the register 3. Memory contents are to be referred to in a similar fashion; for instance, M [8] refers to the contents of memory at location 8 (or the 8th memory cell).

Since memory is organized into cells of 1 byte, whereas the memory word size is 2 bytes, two adjacent memory cells together make up a memory word. So, memory word at the memory address 8 would be defined as 1 byte at address 8 and 1 byte at address 9. To refer to 16-bit memory words, we make use of a special notation, the concatenation of two memory locations. Therefore, to refer to the 16-bit memory word at location 8, we would write M[8]©M[9]. As we employ the big-endian format,

$$M[8] \langle 15 \dots 0 \rangle := M[8] \text{©} M[9]$$

So in our notation © is used to represent concatenation.

Little endian puts the smallest numbered byte at the least-significant position in a word, whereas in big endian, we place the largest numbered byte at the most significant position. Note that in our case, we use the big-endian convention of ordering bytes. However, within each byte itself, the ordering of the bits is little endian.

FALCON-A Features

The FALCON-A processor has fixed-length instructions, each 16 bits (2 bytes) long. Addressing modes supported are limited, and memory is accessed through the load/store instructions only.

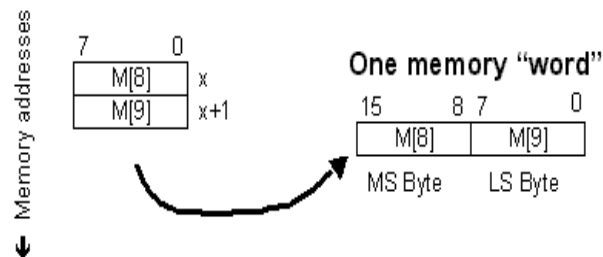
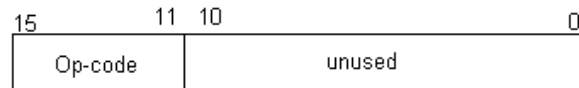


Fig. Big- Endian Notation

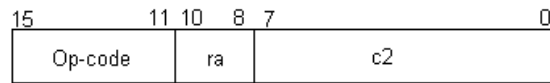
FALCON-A Instruction Formats

Three categories of instructions are going to be supported by the FALCON-A processor; arithmetic, control, and data transfer instructions. Arithmetic instructions enable mathematical computations. Control instructions help change the flow of the program as and when required. Data transfer operations move data between the processor and memory. The arithmetic category also includes the logical instructions. Four different types of instruction formats are used to specify these instructions. A brief overview of the various fields in these instructions formats follows.

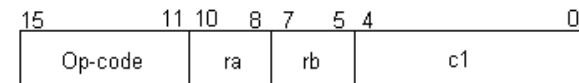
Type I instruction format is shown in the given figure. In it, 5 bits are reserved for the op-code (bits 11 through 15). The rest of the bits are unused in this instruction type, which means they are not considered.



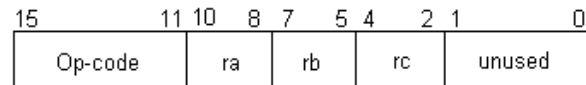
Type II instruction shown in the given figure, has a 5-bit op-code field, a 3-bit register field, and an 8-bit constant (or immediate operand) field.



Type III instructions contain the 5-bit op-code field, two 3-bit register fields for source and destination registers, and an immediate operand field of length 5 bits.



Type IV instructions contain the op-code field, two 3-bit register fields, a constant field on length 3 bits as well as two unused bits. This format is shown in the given figure.



Encoding of registers

We have a register file comprising of eight general-purpose registers in the CPU. To encode these registers in the binary, so they can be referred to in various instructions, we require $\log_2(8) = 3$ bits. Therefore, we have already allocated three bits per register in the instructions, as seen in the various instruction formats. The encoding of registers in the binary format is shown in the given table.

It is important to note here that the register R0 has special usage in some cases. For instance, in load/ store operations, if register R0 is used as a second operand, its value is considered to be zero. R0 has special usage in the multiply and divide (mul & div) instructions as well.

Registers	Encoding
R0	000
R1	001
R2	010
R3	011
R4	100
R5	101
R6	110
R7	111

Fig. Register Encodings

Instructions and instruction formats

We return to our discussion of instruction formats in this section. We will now classify which instructions belong to what instruction format types.

Type I

Five of the instructions included in the instruction set of FALCON-A belong to type I instruction format. These are

1. nop (op-code = 21)
This instruction is to instruct the processor to 'do nothing', or, in other words, do 'no operation'. This instruction is generally useful in pipelining. We will study pipelining later in the course.
2. reset (op-code = 30)
3. halt (op-code=31)
4. int (opcode= 26)
5. iret (op-code= 27)

All of these instructions take no operands, therefore, besides the 5 bits used for the op-code, the rest of the bits are unused.

Type II

There are nine FALCON-A instructions that belong to this type. These are listed below.

1. movi (op-code = 7)
The movi instruction loads a register with the constant (or the immediate value) specified as the second operand. An example is
`movi R3, 56 R[3] ← 56`
This means that the register R3 will have the value 56 stored in it as this instruction is executed.
2. in (op-code = 24)
This instruction is to load the specified register from input device. An example and its interpretation in register transfer language are
`in R3, 57 R [3] ← IO [57]`
3. out (op-code = 25)
The 'out' instruction will move data from the register to the output device specified in the instruction, as the example demonstrates:
`out R7, 34 IO [34] ← R [7]`
4. ret (op-code=23)
This instruction is to return control from a subroutine. This is done using a register, where the return address is stored. As shown in the example, to return control, the program counter is assigned the contents of the register.
`ret R3 PC ← R [3]`
5. jz (op-code= 19)
When this instruction is executed, the value of the register specified in the field ra is checked, and if it is equal to zero, the Program Counter is advanced by the jump(value) specified in the instruction.
`jz r3, [4] (R[3]=0): PC← PC+ 4;`
In this example, register r3's value is checked, and if found to be zero, PC is advanced by 4.
6. jnz (op-code= 18) This instruction is the reverse of the jz instruction, i.e., the jump (or the branch) is taken, if the contents of the register specified are not equal to zero.

jnz r4, [variable] (R[4]≠0): PC← PC+ variable;

7. jpl (op-code= 16) In this instruction, the value contained in the register specified in the field ra is checked, and if it is positive, the jump is taken.

jpl r3, [label] (R[3]≥0): PC ← PC+ (label-PC);

8. jmi (op-code= 17) In this case, PC is advanced (jump/branch is taken) if the register value is negative

jmi r7, [address] (R[7]<0): PC← PC+ address;

Note that, in all the instructions for jump, the jump can be specified by a constant, a variable, a label or an address (that holds the value by which the PC is to be advanced).

A variable can be defined through the use of the '.equ' directive. An address (of data) can be specified using the directive '.db' or '.dw'. A label can be specified with any instruction. In its usage, we follow the label by a colon ':' before the instruction itself. For example, the following is an instruction that has a label 'alfa' attached to it

alfa: movi r3 r4

Labels implement relative jumps, 128 locations backwards or 127 locations forward (relative to the current position of program control, i.e. the value in the program counter). The compiler handles the interpretation of the field c2 as a constant/ variable/ label/ address. The machine code just contains an 8-bit constant that is added to the program counter at run-time.

9. jump (op-code= 20)

This instruction instructs the processor to advance the program counter by the displacement specified, unconditionally (an unconditional jump). The assembler allows the displacement (or the jump) to be specified in any of the following ways

jump [ra + constant]
jump [ra + variable]
jump [ra + address]
jump [ra + label]

The types of unconditional jumps that are possible are

- Direct
- Indirect
- PC relative (a 'near' jump)
- Register relative (a 'far' jump)

The c2 field may be a constant, variable, an address or a label.

A direct jump is specified by a PC-label.

An indirect jump is implemented by using the C2 field as a variable.

In all of the above instructions, if the value of the register ra is zero, then the Program Counter is incremented (or decremented) by the sign-extended value of the constant specified in the instruction. This is called the PC-relative jump, or the 'near' jump. It is denoted in RTL as:

(ra=0):PC← PC+(8αC2<7>)@C2<7..0>;

If the register ra field is non-zero, then the Program Counter is assigned the sum of the sign-extended constant and the value of register specified in the field ra. This is known as the register-relative, or the 'far' jump. In RTL, this is denoted as:

(ra≠0):PC← R[ra]+(8αC2<7>)@C2<7..0>;

Note that C2 is computed by sign extending the constant, variable, address, or (label – PC). Since we have 8 bits available for the C2 field (which can be a constant, variable, address or a PC-label), the range for the field is -128 to + 127. Also note that the compiler does not allow an instruction with a negative sign before the register name, such as ‘jump [-r2]’. If the C2 field is being used as an address, it should always be an even value for the jump instruction. This is because our instruction word size is 16 bits, whereas in instruction memory, the instruction memory cells are of 8 bits each. Two consecutive cells together make an instruction.

Type III

There are nine instructions of the FALCON-A that belong to Type III. These are:

1. **andi** (op-code = 9)
 The **andi** instruction bit-wise ‘ands’ the constant specified in the instruction with the value stored in the register specified in the second operand register and stores the result in the destination register. An example is:
`andi r4, r3, 5`
 This instruction will bit-wise and the constant 5 and R[3], and assign the value thus obtained to the register R[4], as given .

$$R[4] \leftarrow R[3] \& 5$$
2. **addi** (op-code = 1)
 This instruction is to add a constant value to a register; the result is stored in a destination register. An example:
`addi r4, r3, 4` $R[4] \leftarrow R[3] + 4$
3. **subi** (op-code = 3)
 The **subi** instruction will subtract the specified constant from the value stored in a source register, and store to the destination register. An example follows.
`subi r5, r7, 9` $R[5] \leftarrow R[7] - 9$
4. **ori** (op-code= 11)
 Similar to the **andi** instruction, the **ori** instruction bit-wise ‘ors’ a constant with a value stored in the source register, and assigns it to the destination register. The following instruction is an example.
`ori r4, r7, 3` $R[4] \leftarrow R[7] \sim 3$
5. **shifl** (op-code = 12)
 This instruction shifts the value stored in the source register (which is the second operand), and shifts the bits left as many times as is specified by the third operand, the constant value. For instance, in the instruction
`shifl r4, r3, 7`
 The contents of the register are shifted left 7 times, and the resulting number is assigned to the register r4.
6. **shiftr** (op-code = 13)
 This instruction shifts to the right the value stored in a register. An example is:
`shiftr r4, r3, 9`
7. **asr** (op-code = 15)
 An arithmetic shift right is an operation that shifts a signed binary number stored in the source register (which is specified by the second operand), to the right, while leaving the sign-bit unchanged. A single shift has the effect of dividing the number by 2. As the number is shifted as many times as is specified in the instruction through the constant value, the binary number of the source register gets divided by the constant value times 2. An example is

asr r1, r2, 5

This instruction, when executed, will divide the value stored in r2 by 10, and assign the result to the register r1.

8. load (op-code= 29)

This instruction is to load a register from the memory. For instance, the instruction

load r1, [r4 +15]

will add the constant 15 to the value stored in the register r4, access the memory location that corresponds to the number thus resulting, and assign the memory contents of this location to the register r1; this is denoted in RTL by:

$$R[1] \leftarrow M[R[4]+15]$$

9. store (op-code= 28)

This instruction is to store a value in the register to a particular memory location. In the example:

store r6, [r7+13]

The contents of the register r6 are being stored to the memory location that corresponds to the sum of the constant 13 and the value stored in the register r7.

$$M[R[7]+13] \leftarrow R[6]$$

Type III Modified

There are 3 instructions in the modified form of the Type III instructions. In the modified Type III instructions, the field c1 is unused.

1. mov (op-code = 6)

This instruction will move (copy) data of a source register to a destination register. For instance, in the following example, the contents of the register r3 are copied to the register r4.

mov r4, r3

In RTL, this can be represented as

$$R[4] \leftarrow R[3]$$

2. not (op-code = 14)

This instruction inverts the contents of the source register, and assigns the value thus obtained to the destination register. In the following example, the contents of register r2 are inverted and assigned to register r4.

not r4, r2

In RTL:

$$R[4] \leftarrow !R[2]$$

3. call (op-code = 22)

Procedure calls are often encountered in programming languages. To add support for procedure (or subroutine) calls, the instruction call is used. This instruction first stores the return address in a register and then assigns the Program Counter a new value (that specifies the address of the subroutine). Following is an example of the call instruction

call r4, r3

This instruction saves the current contents (the return address) of the Program Counter into the register r4 and assigns the new value to the PC from register r3.

$$R[4] \leftarrow PC, PC \leftarrow R[3]$$

Type IV

Six instructions belong to the instruction format Type IV. These are

1. add (op-code = 0)

This instruction adds contents of a register to those of another register, and assigns to the destination register. An example:

and r4, r3, r5
 $R[4] \leftarrow R[3] + R[5]$

2. sub (op-code = 2)

This instruction subtracts value of a register from another the value stored in another register, and assigns to the destination register. For example,

sub r4, r3, r5
In RTL, this is denoted by
 $R[4] \leftarrow R[3] - R[5]$

3. mul (op-code = 4)

The multiply instruction will store the product of two register values, and stores in the destination register. An example is

mul r5, r7, r1

The RTL notation for this instruction will be

$R[0] \odot R[5] \leftarrow R[7] * R[1]$

4. div (op-code= 5)

This instruction will divide the value of the register that is the second operand, by the number in the register specified by the third operand, and assign the result to the destination register.

div r4, r7, r2 $R[4] \leftarrow R[0] \odot R[7] / R[2], R[0] \leftarrow R[0] \odot R[7] \% R[2]$

5. and (op-code= 8)

This 'and' instruction will obtain a bit-wise 'and' of the values of two registers and assigns it to a destination register. For instance, in the following example, contents of register r4 and r5 are bit-wise 'anded' and the result is assigned to the register r1.

and r1, r4, r5

In RTL we may write this as

$R[1] \leftarrow R[4] \& R[5]$

6. or (op-code= 10)

To bit-wise 'or' the contents of two registers, this instruction is used. For instance,

or r6, r7, r2

In RTL this is denoted as

$R[6] \leftarrow R[7] \sim R[2]$

FALCON-A: Instruction Set Summary

We have looked at the various types of instruction formats for the FALCON-A, as well as the instructions that belong to each of these instruction format types. In this section, we have simply listed the instructions on the basis of their functional groups; this means that the instructions that perform similar class of operations have been listed together.

Data Transfer Instructions	Mnemonic	opcode
move	mov	00110 (6)
Move immediate	movi	00111 (7)
Input to register	in	11000 (24)
Output from register	out	11001 (25)
Load from memory	load	11101 (29)
Store into memory	store	11100 (28)

Fig. Data Transfer Instructions

jump instruction	Mnemonic	opcode
jump if positive	jpl	10000 (16)
jump if negative	jmi	10001 (17)
jump if not zero	jnz	10010 (18)
jump if zero	jz	10011 (19)
jump	jump	10100 (20)

Fig. Jump Instructions

Control Instruction	Mnemonic	opcode
No operation	nop	10101 (21)
call	call	10110 (22)
return	ret	10111 (23)
interrupt	int	11010 (26)
Interrupt return	iret	11011 (27)
reset	reset	11110 (30)
halt	halt	11111 (31)

Fig. Control Instructions

Examples for FALCON-A

In this section we take up a few sample problems related to the FALCON-A processor. This will enhance our understanding of the FALCON-A processor, as well as of the general concepts related to general processors and their instruction set architectures. The problems we will look at include

1. Identification of the instruction types and operands
2. Addressing modes and RTL description
3. Branch condition and status of the PC
4. Binary encoding for instructions

Example 1:

Identify the types of given FALCON-A instructions and specify the values in the fields

Instruction	Type	ra	rb	rc	c1	c2
movi r1, 2						
add r1, r2, r3						
nop						
load r2, [r5 + 6]						
jz r0, [3]						

Fig. Example 1

Solution

The solution to this problem is quite straightforward. The types of these instructions, as well as the fields, have already been discussed in the preceding sections.

Instruction	Type	ra	rb	rc	c1	c2
movi r1, 2	II	r1	-	-	-	2
add r1,r2,r3	IV	r1	r2	r3	-	-
nop	I	-	-	-	-	-
load r2,[r5 + 6]	III	r2	r5	-	6	-
jz r0, [3]	II	r0	-	-	-	3

Fig. Solution 1

We can also find the machine code for these instructions. The machine code (in the hexadecimal representation) is given for these instructions in the given table.

Instruction	Machine Code	ra	rb	rc	c1	c2
movi r1, 2	3902h	r1	-	-	-	2
add r1,r2,r3	014Ch	r1	r2	r3	-	-
nop	A800h	-	-	-	-	-
load r2,[r5 + 6]	EAA6h	r2	r5	-	6	-
jz r0, [3]	9803h	r0	-	-	-	3

Fig. Machine Code

Example 2:

Identify the addressing modes and Register Transfer Language (RTL) description (meaning) for the given FALCON-A instructions

Instruction	Addressing mode	RTL description (meaning)
load r2,[r4 + 8]		
jnz r1,[54]		
shiftl r1,r2,4		
addi r3,r6,2		
sub r1, r7,r2		

Fig. Example 2

Solution

Addressing modes relate to the way architectures specify the address of the objects they access. These objects may be constants and registers, in addition to memory locations.

Instruction	Addressing mode	RTL description (meaning)
load r2,[r4 + 8]	Displacement	$R[2] \leftarrow M[R[4]+8]$
jnz r1, [54]	Relative	$(R[1] \neq 0):$ $PC \leftarrow PC+54$
shiftl r1,r2,4	Immediate	Shift r2 left 4 times and store in r1
addi r3,r6,2	Immediate	$R[3] \leftarrow R[6]+2$
sub r1, r7,r2	Register	$R[1] \leftarrow R[7]-R[2]$

Fig. Solution 2

Example 3: Specify the condition for the branch instruction and the status of the PC after the branch instruction executes with a true branch condition

Instruction	Condition	PC status
jz r2,[35]		
jump [12]		
jnz r6, [3]		
jp1 r1, [45]		
jmi r2, [20]		

Fig. Example 3

Solution

We have looked at the various jump instructions in our study of the FALCON-A. Using that knowledge, this problem can be solved easily.

Instruction	Condition	PC status
jz r2,[35]	If R[2]==0	PC \leftarrow PC+35
jump [12]	always	PC \leftarrow PC+12
jnz r6, [3]	If R[6] \neq 0	PC \leftarrow PC+3
jp1 r1, [45]	If R[1] \geq 0	PC \leftarrow PC+45
jmi r2, [20]	If R[2] < 0	PC \leftarrow PC+20

Fig. Solution 3

Example 4: Specify the binary encoding of the different fields in the given FALCON-A instructions.

Instruction	TYPE	opcode	ra	rb	rc	C1(5 bits) OR C2(8 bits)
store r4, [r1+8]						
sub r3,r6,r5						
shiftr r4,r6,9						
jump [10]						
halt						

Fig. Example 4

Solution

We can solve this problem by referring back to our discussion of the instruction format types. The op-codes for each of the instructions can also be looked up from the tables. ra, rb and rc (where applicable) registers' values are obtained from the register encoding table we looked at. The constants C1 and C2 are there in instruction type III and II respectively. The immediate constant specified in the instruction can also be simply converted to binary, as shown.

Instruction	TYPE	opcode	ra	rb	rc	C1(5 bits) OR C2(8 bits)
store r4, [r1+8]	III	11100	100	001	-	01000
sub r3,r6,r5	IV	00010	011	110	101	-
shiftr r4,r6,9	III	01101	100	110	-	01001
jump [10]	II	10100	-	-	-	0000 1010
halt	I	11111	-	-	-	-

Fig. Solution 4