marca - McAdam's RISC Computer Architecture Implementation Details

Wolfgang Puffitsch

February 2, 2007

1 General

- 16 16-bit registers
- 16KB instruction ROM (8192 instructions)
- 8KB data RAM
- 256 byte data ROM
- 75 instructions
- 16 interrupt vectors

2 Internals

The processor features a 4-stage pipeline:

- instruction fetch
- instruction decode
- execution/memory access
- write back

This scheme is similar to the one used in the MIPS architecture, only execution and write back stage are drawn together. For our architecture does not support indexed addressing, it does not need the ALU's result and can work in parallel, having the advantage of reducing the possible hazards.

Figure 1 shows a rough scheme of the internals of the processor.

2.1 Branches

Branches are not predicted and if executed they stall the the pipeline, leading to a total execution time of 4 cycles. The fetch stage is not stalled, the decode stage however is stalled for two cycles to compensate that.

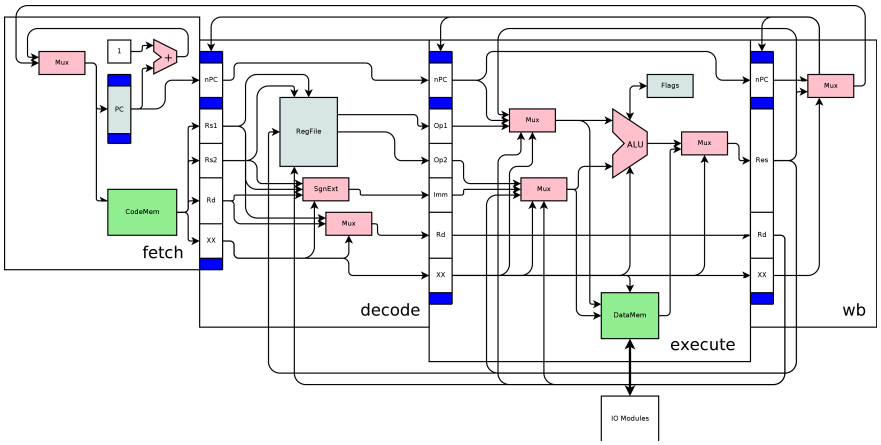

Figure 1: Internal scheme

2.2 Instruction fetch

This stage is not spectacular: it simply reads an instruction from the instruction ROM, and extracts the bits for the source and destination registers.

2.3 Instruction decode

This stage translates the bit-patterns of the opcodes to the signals used internally for the operations. It also holds the register file and handles access to it. Immediate values are also constructed here.

2.4 Execution / Memory access

The execution stage is the heart and soul of the processor: it holds the ALU, the memory/IO unit and a unit for interrupt handling.

2.4.1 ALU

The ALU does all arithmetic and logic computations as well as taking care of the processors flags (which are organized as seen in table 1).

Bit 15															Bit 0
										P	I	N	V	C	Z

Table 1: The flag register

Operations which need more than one cycle to execute (multiplication, division and modulo) block the rest of the processor until they are finished.

2.4.2 Memory/IO unit

The memory/IO unit takes care of the ordinary data memory, the data ROM (which is mapped to the addresses right above the RAM) and the communication

to peripheral modules. Peripheral modules are located within the memory/IO unit and mapped to the highest addresses.

The memories (the instruction ROM too) are Altera specific; we decided not to use generic memories, because *Quartus* can update the contents of its proprietary ROMs without synthesizing the whole design. Because all memories are single-ported (and thus fairly simple) it should be easy to replace them with memories specific to other vendors.

We also decided against the use of external memories; larger FPGAs can accommodate all addressable memory on-chip, so the implementation overhead would not have paid off.

Accesses which take more than one cycle (stores to peripheral modules and all load operations) block the rest of the processor until they are finished.

Peripheral modules The peripheral modules use a slightly modified version of the SimpCon interface. The SimpCon specific signals are pulled together to records, and the words which can be read/written are limited to 16 bits. For accessing such a module, one may only use `load` and `store` instructions which point to aligned addresses.

UART The built-in UART is derived from the sc_uart from Martin Schöberl. Apart from adapting the SimpCon interface, an interrupt line and two bits for enabling/masking receive (bit 3 in the status register) and transmit (bit 2) interrupts. In the current version address 0xFFF8 (-8) correspond to the UART's status register and address 0xFFFA (-6) to the wr_data/rd_data register.

2.4.3 Interrupt unit

The interrupt unit takes care of the interrupt vectors and, of course, the triggering of interrupts. Interrupts are executed only if the global interrupt flag is set, none of the other units is busy and the instruction in the execution stage is valid (it takes 3 cycles after jumps, branches etc. until a new valid instruction is in that stage).

Instructions which cannot be decoded as well as the "error" instruction trigger interrupt 0; the ALU can trigger interrupt 1 (division by zero), the memory unit can trigger interrupt 2 (invalid memory access). In contrast to all other interrupts, these three interrupts do not repeat the instruction which is executed when they occur.

2.5 Write back

The write back stage passes on the result of the execution stage to all other stages.

3 Assembler

The assembler *spar* (SPear Assembler Recycled) uses a syntax quite like usual Unix-style assemblers. It accepts the pseudo-ops `.file`, `.text`, `.data`, `.bss`, `.align`, `.comm`, `.lcomm`, `.org` and `.skip` with the usual meanings. The mnemonic `data` initializes a byte to some constant value. In difference to the instruction

set architecture specification, `mod` and `umod` accept three operands (if a move is needed, it is silently inserted).

The assembler produces three files: one file for the instruction ROM, one file for the even bytes of the data ROM and one file for the odd bytes of the instruction ROM. The splitting of the data is necessary, because the data memories internally are split into two 8-bit memories in order to support unaligned memory accesses without delays.

Three output formats are supported: .mif (Memory Initialization Format), .hex (Intel Hex Format) and a binary format designed for download via UART.

4 Resource usage and speed

The processor was synthesized with *Quartus II* for the *Cyclone EP1C12Q240C8* FPGA with 12060 logic cells and 29952 bytes of on-chip memory available.

The processor needs ∼3550 logic cells or 29% when being compiled for maximum clock frequency, which is ∼60 MHz. When optimizing for area, it needs ∼2600 logic cells or 22% at ∼25 MHz.

The processor uses 24832 bytes or 83% of on-chip memory.

5 Example

5.1 Reversing a line

In listing 1 one can see how to interface the uart via interrupts. The program reads in a line from the UART and the writes it back reversed. The lines 1 to 4 show how to instantiate memory (the two bytes defined form the DOS-style end-of-line). The lines 7 to 25 initialize the registers and register the interrupt vectors, line 28 builds a barrier against the rest of the code.

The lines 32 to 76 form the interrupt service routine. It first checks if it is operating in read or in write mode. When reading, it reads from the UART and stores the result. A mode switch occurs when a newline character is encountered. In write mode the contents of the buffer is written to the UART and switching back to read mode is done when finished.

In figure 2 the results of the simulation are presented.

Listing 1: Example for the UART and interrupts

```
1  .data
2          data 0x0A
3          data 0x0D
4  buffer:
5
6  .text
7  ;;; initialization
8          ldib    r0, -8          ; config/status
9          ldib    r1, -6          ; data
10
11         ldil    r2,  lo(buffer) ; buffer address
12         ldih    r2,  hi(buffer) ; buffer address
13
14         ldib    r3,  0x0A       ; newline character
15         ldib    r4,  0x0D       ; carriage return
16
```

```
17          ldib    r5,  0          ; mode
18
19          ldib    r7,  isr        ; register isr
20          stvec   r7,  3
21
22          ldib    r7, (1 << 3)    ; enable receive interrupts
23          store   r7,  r0
24
25          sei                     ; enable interrupts
26
27  ;;; loop forever
28  loop:   br loop
29
30
31  ;;; ISR
32  isr:
33          cmpi    r5, 0           ; check mode
34          brnz    write_mode
35
36  ;;; reading
37  read_mode:
38          load    r7, r1          ; read data
39
40          cmp     r7, r3          ; change mode upon newline
41          brnz    read_CR
42
43          ldib    r7, (1 << 2)    ; do the change
44          store   r7, r0
45          ldib    r5, 1
46          reti
47
48  read_CR:
49          cmp     r7, r4          ; ignore carriage return
50          brnz    read_cont
51          reti
52
53  read_cont:
54          storel  r7, r2          ; store date
55          addi    r2,  1
56          reti
57
58  ;;; writing
59  write_mode:
60          addi    r2, -1
61
62          cmpi    r2, -1          ; change mode if there is no more data
63          brnz    write_cont
64
65          ldil    r2,  lo(buffer) ; correct pointer to buffer
66          ldih    r2,  hi(buffer)
67
68          ldib    r7, (1 << 3)    ; do the change
69          store   r7, r0
70          ldib    r5, 0
71          reti
72
73  write_cont:
74          loadl   r7, r2          ; write data
75          store   r7, r1
76          reti
```

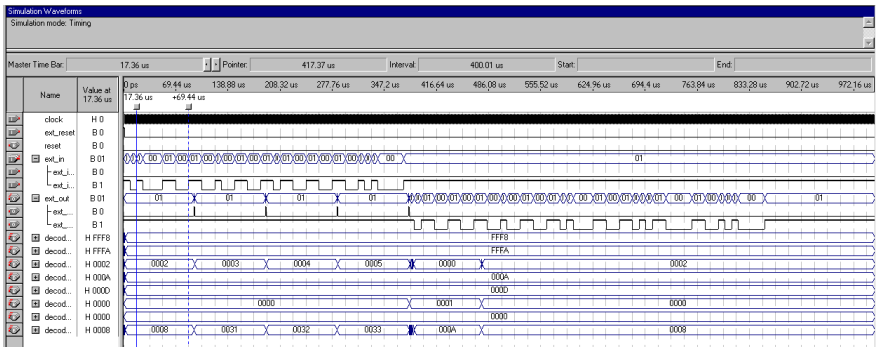

Figure 2: Simulation results

5.2 Computing factorials

The example in 2 computes the factorials of $1 \ldots 9$ and writes the results to the PC via UART. Note that the last result transmitted will be wrong, because it is truncated to 16 bits.

Listing 2: Computing factorials

```
1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;;; factorial
3  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
4  ;;; compute factorials of 1 to 9 and write results to
5  ;;; the PC via UART
6  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
7
8  .data
9
10 ;;; the numbers to be written are placed here
11 iobuf:
12         data 0x0A
13         data 0x0D
14         data 0
15         data 0
16         data 0
17         data 0
18         data 0
19         data 0
20
21 ;;; stack for recursive calls of factorial()
22 stack:
23
24 .text
25 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
26 ;;; main()
27 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
28         ldib    r15, 1          ; number to start
29         ldib    r5, 10          ; number to stop
30
31         ldil    r1, lo(stack)   ; setup for factorial()
32         ldih    r1, hi(stack)
33         ldil    r2, lo(factorial)
34         ldih    r2, hi(factorial)
35
36         ldib    r6, 0x30        ; setup for convert()
37         ldib    r7, 10
```

```
38          ldil    r8, lo(iobuf)
39          ldih    r8, hi(iobuf)
40          ldil    r9, lo(convert)
41          ldih    r9, hi(convert)
42
43          ldib    r12, -8         ; enable write interrupts
44          ldib    r11, (1 << 2)
45          store   r11, r12
46
47          ldil    r12, lo(isr)    ; register isr() to be called upon
48          ldih    r12, hi(isr)    ; interrupt #3
49          stvec   r12, 3
50
51          ldib    r12, -6         ; address where to write data
52                                  ; to the UART
53
54  loop:
55          mov     r0, r15         ; r0 is the argument
56          call    r2, r3          ; call factorial()
57          call    r9, r3          ; call convert()
58
59  wait:   getfl   r13
60          btest   r13, 4          ; interrupts still enabled?
61          brnz    wait
62
63          addi    r15, 1          ; loop
64          cmp     r15, r5
65          brnz    loop
66
67  exit:   br      exit            ; stop here after all
68
69  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
70  ;;; converting content of r4 to a string
71  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
72  convert:
73          addi    r8, 2
74  convert_loop:
75          umod    r4, r7, r10     ; the conversion
76          add     r10, r6, r10
77          storel  r10, r8
78          addi    r8, 1
79
80          udiv    r4, r7, r4      ; next digit
81
82          cmpi    r4, 0
83          brnz    convert_loop
84
85          sei                     ; trigger write
86          jmp     r3
87
88  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
89  ;;; write out content of iobuf
90  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
91  isr:
92          cmpi    r8, iobuf       ; reached end?
93          brz     written
94
95          addi    r8, -1          ; write data to UART
96          loadb   r10, r8
97          store   r10, r12
98
99          reti
```

```
100
101 written:
102         getshfl r10
103         bclr    r10, 4          ; clear interrupt flag
104         setshfl r10
105         reti
106
107 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
108 ;;; recursively compute factorial
109 ;;; argument:           r0
110 ;;; return value:       r4
111 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
112 factorial:
113         cmpi    r0, 1           ; reached end?
114         brule   fact_leaf
115
116         store   r0, r1          ; push argument and return
117         addi    r1,  2          ; address onto stack
118         store   r3, r1
119         addi    r1,  2
120
121         addi    r0, -1          ; call factorial(r0-1)
122         call    r2, r3
123
124         addi    r1, -2          ; pop argument and return
125         load    r3, r1          ; address from stack
126         addi    r1, -2
127         load    r0, r1
128
129         mul     r0, r4, r4      ; return r0*factorial(r0-1)
130         jmp     r3
131
132 fact_leaf:                      ; factorial(1) = 1
133         ldib    r4, 1
134         jmp     r3
```

6 Versions Of This Document

2006-12-14: Draft version **0.1**
2006-12-29: Draft version **0.2**

- A few refinements.

2007-01-22: Draft version **0.3**

- Added another example.

2007-02-02: Draft version **0.4**

- Updated resource usage and speed section.