RETURN

Build your own RISC processor simulator

Vijaya Sagar Vinnakota (vijaya.sagar@wipro.com)
Wipro Technologies

Abstract

This tutorial paper presents a way to design a RISC simulator in software. Design concepts and sample 'C' implementation excerpts are shown to support the concepts.

The tutorial does not assume prior experience in programming a RISC processor. An understanding and appreciation of the RISC philosophy is desirable, yet, not mandatory.

Keywords

RISC, CISC, pipeline, inter-locking, profiling

Introduction

Before we plunge into designing a RISC processor simulator, let us spend a few moments in understanding the philosophical differences between the traditional RISC and CISC architectures.

A quick look at the evolution of automated computing indicates a continuous improvement in the ease of programming vis-à-vis program complexity. This can primarily be attributed to the advent of 'high-level' programming languages, which liberated the computer programmer from writing code[1] directly in machine or assembler languages.

High-level programming languages are programmer-friendly but they introduce an unavoidable overhead of a multi-stage translation. A program in a typical high level programming language goes through a series of complex transformations such as lexical analysis, parsing, code generation and optimisation before its equivalent machine code is produced[2].

Obviously, such a translation would be entirely unnecessary if the target processor directly interprets the constructs of the programming language. However, such an option is not scaleable[3] as the programming languages continue to grow in number as well as complexity. Imagine a system that needs an additional processor for every new programming language designed! It is precisely this problem the program translators address.

The reign of CISC

Even as we rule out language specific processors as a solution, it seems

---

[1] No wonder the name 'code' was chosen for programs, in the machine-language programming days

[2] These are apart from incidental processes such as adding debug information and linking which do not alter the fundamental transform

[3] Very much possible nonetheless as demonstrated by the good old Burroughs LISP machine!

desirable to have a good match between constructs of programming languages and instruction sets of generic processors. This makes program translation easier as the code generator only needs to establish a mapping between the language constructs and the target processor's instruction set. This, in a nutshell, is the CISC (Complex Instruction Set Computers) philosophy.

The CISC philosophy reigned unchallenged for well over a decade in which it had as popular ambassadors, microprocessors from IBM, Digital, Motorola, Intel, Zilog and the like. These microprocessors supported a large number of addressing modes, varied data processing instructions including complicated mathematical operations.

However, the CISC lunch did not come free. Complex micro-code (which consumed a significant portion of silicon on the real-estate starved chip), multi-cycle instructions (which blocked interrupts as long as they were executing!) and a performance that was slower as compared to memory offered by new technologies made CISC a good candidate for second choice. Only, the first choice was non-existent.

The emergence of RISC

As the semiconductor industry strove hard to produce all-new and highly improved mousetraps[4], Ditzel, Hennessy and Patterson made a case for reinventing the whole wheel based on the principles of, what they termed, a "reduced instruction set computer" architecture. Some students at Berkeley took the RISC idea seriously and designed a simple processor by name 'RISC I'. This marked the beginning of the end of pure-CISC era.

[4] Read CISC micro-processors

The RISC architecture was based on a simple observation that most programs spent most of their time (over 70%) in moving data between the processor and memory, and in making program control decisions based on that data. RISC designers devoted most of their attention towards making processors that run faster for these frequently used instructions. These processors had their functional units (ALU, register files, decode logic, control signal logic etc.) organised so as to be operated in parallel. This allowed for splitting the lifecycle of each instruction into multiple phases, with one functional unit per phase. This approach, known as 'pipelining' closely resembles the assembly line in a manufacturing industry. Fig.1 is a snapshot of on such three-stage instruction pipeline. The lightly shaded portion shows the stages that would be executed in the future while the darkly shaded portion shows those that are completed.

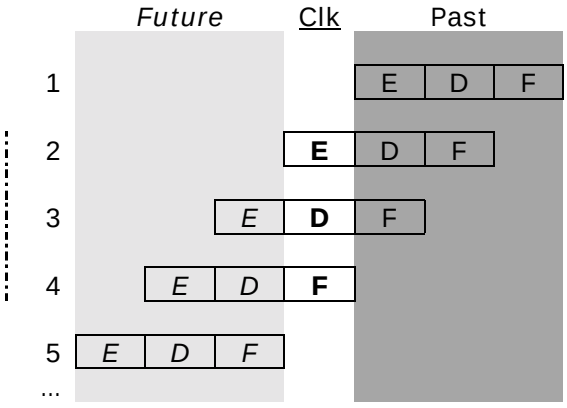

Fig.1 – Snapshot of an instruction pipeline
F: Fetch D: Decode E: Execute

Each instruction enters the pipeline from the memory retires after completing execution. Of the five instructions shown in the figure, only three are in the pipeline, in various stages of execution. The first instruction has completed its three stages and has retired. The fifth

instruction is in the memory[5] and is yet to enter the pipeline.

In the current clock cycle (shown as 'Clk' in the figure), the processor parallely executes the E-stage of instruction-2, D-stage of 3 and F of 4. This is made possible by a design, which ensures that no two active stages require the same processor functional unit simultaneously (i.e., in the same clock cycle).

Throughput and turnaround time:

It is common to hear and read that most RISC processor instructions are single-cycle in nature. This confuses first time RISCers as it contradicts their understanding of the processor pipeline.

The confusion can be reduced by re-reading such statements as: "Most RISC processor instructions take one clock cycle *per pipeline-stage*". Alternately, the 'single-cycle' can be interpreted as referring to *a single pipeline-cycle* instead of being seen as a single clock-cycle.

The fact is that all instructions are multi-cycle in nature. Every instruction takes at least as many clock cycles to complete, as the number of pipeline stages. This is a measure of the instruction *turnaround time*.

However, setting special conditions aside, a pipeline has a *throughput* of one instruction per clock-cycle, once it reaches steady state. Fig.2 through Fig.5 delineate these two concepts.

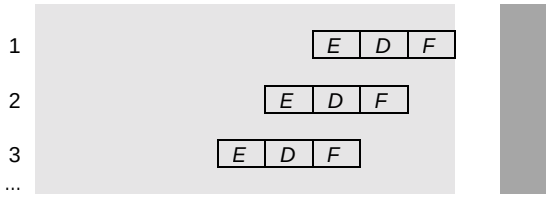

Fig.2 Initial state of a 3-stage pipeline

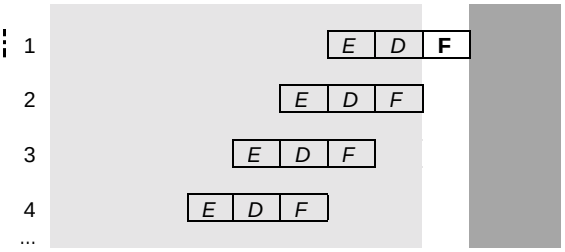

Fig.3 One clock-cycle past the initial state

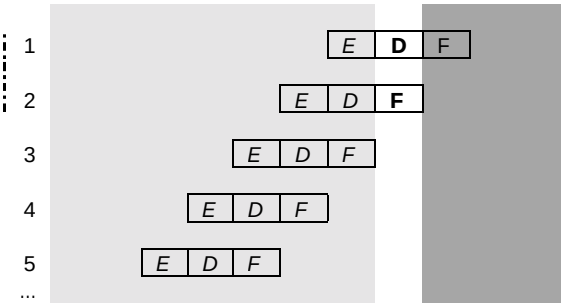

Fig.4 Two clock-cycles past the initial state

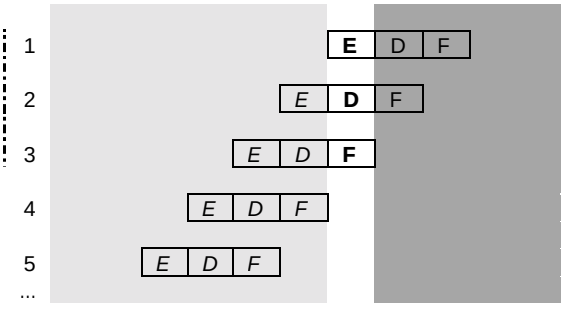

Fig.5 Three clock-cycles past the initial state

As can be seen, it takes three clock cycles for this pipeline to get 'filled'. At the end of the third cycle, the first instruction completes execution and retires. With this the pipeline reaches a steady state as shown in Fig.1. From

[5] Anywhere in the memory hierarchy

thereon, one instruction retires per clock-cycle.

While the ideal number of stages for a pipeline is debatable, a few limiting factors help make this decision easier for a RISC processor designer:

- The number of parallely usable functional units (including internal buses!)
- The number of frequently used instructions that take more than one clock-cycle to complete any of their stages (e.g., memory load/store, multiplication, branch instructions)
- Non-interlocking *vs.* stalling pipeline approach[6]

Processor Simulation

The process of design, development and testing of a processor takes a long time during which many models are made to fine-tune its functionality and performance, before the production is commenced. These models simulate the processor behaviour in various levels of detail. For instance, typical FPGA models match their processor's functionality but not the timing characteristics. Yet, these models help the designers identify and correct most of the flaws.

The production of hardware models is usually discontinued after the processor is proven and accepted in the market. However, the software models, also more popularly known as simulators, continue to be used, enhanced and produced as long as the processor is in use. In spite of certain limitations (such as being unable to exactly reproduce time critical behaviour such as interrupt latency and bus cycles), these simulators serve as close functional approximations and inexpensive alternatives to their processors, the reference hardware boards and associated environment.

Design Considerations

It is fairly trivial to design a processor simulator as a simple transformation function / mapping between the processor's instruction set – I_p and the instruction set – I_h of the simulator's host machine. This mapping may simply be based on a lookup-table if I_p is a functional subset of I_h i.e., if there is a one-one mapping between I_p and I_h (with allowance to difference in instruction formats). If the two instruction sets are significantly different from each other, a slightly involved mapping has to be employed. In this case, each instruction of I_p has to be implemented in terms of two or more instructions from I_h.

These mappings can be implemented by designing the simulator as an interpreter for the instruction stream of a program written for the target processor. The simulator can take as input, either the executable instructions of Ip or their assembler mnemonics. In either case, the interpretation is easier by using an intermediate high-level language – HL that is supported by the host. The translation from HL to Ih is best left to the host's HL translator[7].

Granularity of simulation

The simple instruction mapping approach suffices most normal programming tasks. However, on a closer look, it becomes evident that there is more to simulating a processor than implementing its

[6] For example, the MIPS pipeline design allows delayed-execution while the ARM lets the pipeline stall

[7] Read as compiler/interpreter

instruction set. A fine grain behavioural simulation should involve modeling key functional blocks and macro blocks that make the processor.

Typical blocks that constitute a RISC processor include ALU, instruction decoder, processor control logic, register files, instruction pipeline, barrel shifters, multipliers, write-buffers and internal buses. Depending on the target application / users, a simulator designer has to include models of these blocks into the simulator. For instance, if the simulator is to be used for detailed clock-cycle level profiling, the simulator must include a good model of the instruction pipeline and its clock.

Simulator Components

The rest of this paper presents a detailed behavioural model of *Crisp*[8] - a hypothetical RISC processor. Instead of explaining the architecture of *Crisp* as a separate section, its simulator design is used as a vehicle to introduce the processor and its components.

The key processor components to be modeled are:

0. Clock
1. Memory Interface
2. Execution Unit
3. Arithmetic and Logic Unit
4. Pipeline and parallelism among components

Clock

For a real processor, a clock signal provides the heartbeat. Each instruction takes a pre-designed number of clock cycles to complete. Such a clock is not an essential requirement for building a software model of the processor. Yet, instruction level profiling and fine grain performance analysis of programs will be difficult if such a model makes no provision for a clock. Also, as will be seen later, a model with a clock eases simulating the behaviour of an instruction pipeline.

While the hardware design of a system clock is fairly complicated and involves high precision engineering for the oscillator and phase locked loops for fine-tuning, its software equivalent can be modelled very easily. A system wide counter can act as the clock with its value being updated at appropriate stages of executing each instruction.

It is clear that this behaviour is opposite to that observed on a real processor where the clock drives the instruction execution. However letting the instruction execution phases drive the clock is a good enough approach for a software simulator.

It might be worthwhile to consider using a floating-point value for the clock counter so as to represent half/quarter cycles or any other intermediate points within a clock cycle for very fine grain timing analysis. e.g.,

- RD, WR signals go high/low at set points in a cycle
- data/address buses contain valid data only during a specific portion of the cycle

Crisp receives its clock from an external source such as a PLL.

Memory

Memory is best modelled as an array of data words. A more sophisticated approach would be to model memory as an abstract data type with features such as separate program and data memories, write protection and

[8] Readers will find some similarities between Crisp and ARM

storage heirarchy (TLB, multi-level cache, primary memory, secondary memory etc.).

Registers can be treated as an extension to the memory model. Register files can be supported by a two dimensional array of data words, with one column per register.

Crisp has 15 general-purpose registers named r0 through r14. By convention, r13 is used as the stack pointer and r14 as the link register for procedure calls. r15, a special register, serves as the program counter (instruction pointer). These registers are 32-bit wide.

Execution Unit

The execution unit can be modelled by as a mapping of the instruction set of the processor being modelled to that of the host processor. Or, as a simple translation of the semantics of a model instruction to that of a language construct interpretable on the host processor. e.g.,

Model instruction:
operator operand_1 operand_2

A 'C' translation:
operator(operand_1, operand_2)

Though it seems unnecessary to introduce one more level of indirection between the model instruction and translation in the form of a function call, its utility becomes evident when it is realised that different types of operators might involve different kinds of processor subsystems. e.g.,

add r0, r1
; involves only registers and ALU

add r0, [r1]
; involves registers, memory and ALU

mov r0, 0x10
; involves only registers (instruction
; register and r0)

mov [r0], 0x10
; involves registers and memory

Arithmetic and Logic Unit

ALU operations come next only to memory operations in number, in any typical program. The ALU can also be modelled on lines similar to those of the execution unit. The operators of the processor being modelled are mapped on to those of the host processor or to those of any language understood on the host processor. e.g.,
Model instruction:
add r0, r1

Execution Unit model:
_add(_reg_r0, _reg_r1)

ALU model:
return (_reg_r0 += _reg_r1);

Crisp does not have a multiplier but has a barrel-shifter to perform shifts of length 1-32 in a single cycle. Most of the *Crisp* instructions are in 3-address code format (with unspecified operands filled by an assembler with default values).

Pipeline and parallelism among components

Most modern processors have a 3-6 stage instruction execution pipeline.

A pipeline helps to maximise the utilisation of different components of a processor, which function in parallel and independent of each other (sharing the same clock).

A software model need not simulate parallelism in the real world time. It is necessary and sufficient if various

components of the processor run in parallel with respect to the software clock that is available in the model.

Crisp employs a 3-stage fetch-decode-execute pipeline. The pipeline is clocked at the same speed as the external clock input.

A *Crisp* simulator

In this section, 'C' code fragments of the simulator will be presented along with suitable explanations wherever required. We take a top-down approach for the design and look at non-trivial functionalities in detail. Firstly, the super-structure of the simulator:

```
int   main(int argc, char *argv[])
{
      extern char *progname;

      /* process arguments */
      progname = argv[0];
      /* ... */

      init_sim();       /* initialise Crisp functional blocks */

      /*
       * load the Crisp instruction stream to be executed into memory.
       * argv[1] holds the stream file name.
       */
      program_start = load_program(argv[1]);

      start_Crisp();    /* Crisp starts executing from address 0 */
}
```

Initialisation

```
void  init_sim(void)
{
      init_regs(0);     /* clear (zero) Crisp registers */
      init_memory(0);   /* clear memory accessible to Crisp */

      init_clock();     /* reset the clock counter to zero */
      init_pipeline();  /* setup an [empty] queue of instructions */
}
```

Though it is usual, at startup, to set the registers and memory to zero, it is better to design for a value other than zero too. For instance, in order to understand memory usage patterns, it is useful to initialise the memory to relatively unique patterns such

as 0xbaba and 0xf00dcafe. Hence, *init_regs()* and *init_memory()* take an integer argument.

The pipeline is best modeled as a queue of instructions. New instructions enter the queue at the tail while the completed instructions exit the pipeline from the head. *init_pipeline()* initialises these head and tail indices.

```
void  init_pipeline(void)
{
      _p_head = _p_tail = 0;
}
```

Crisp in action

The main phase of simulation opens with *start_Crisp()*, as the instruction stream execution starts from memory word zero – Crisp's reset vector address.

```
void  start_Crisp(void)
{
      extern int pending_bds; /* see 'Handling branches' */

      set_reg_val(REG_NEXTPC, RESET_VEC_ADDR);
      set_reg_val(REG_PC, get_reg_val(REG_NEXTPC) - 8);
      pending_bds = 0;

      /* pipelined execution */
      while (1) {
            start_new_cycle();
            exec_pipeline_stages();
            retire_instrs();
      }
}
```

All Crisp instructions take exactly three cycles to complete. Each instruction in the pipeline completes one stage of execution, per clock cycle. e*xec_pipeline_stages()* illustrates this. The reason for REG_PC trailing REG_NEXTPC by 8-bytes (two instructions) becomes evident as we go through the inner workings of all the three stages.

```
void  exec_pipeline_stages(void)
{
      _decoder_output cur_decoder_output;

      /*
       * simulate pipelining by retaining a decoded instruction
       * across invocations
```

```
	 */
	static	_decoder_output prev_decoded_instr = {INVALID};

	/*
	 * The three pipeline stages:
	 * 1. fetch the instruction pointed to by PC
	 * 2. decode the instruction trailing the head by one position
	 * 3. execute the instruction which was previously decoded
	 */

	fetch(get_reg_val(REG_NEXTPC));
	decode(peek_pipeline(HEAD, 1), &cur_decoder_output);
	execute(prev_decoded_instr);

	/* prepare for next cycle */
	prev_decoded_instr = cur_decoder_output;
}
```

As the inline comments suggest, *peek_pipeline()* takes as arguments, an enumerated reference position (*HEAD*/*TAIL*) and an offset (*0-2*) from that position (towards the other position). It returns a (possibly NULL) pointer to the required instruction.

Crisp pipeline mechanics

The *fetch()* stage simply requests the memory subsystem for the instruction at the address contained in an internal register *REG_NEXTPC* and puts it into the pipeline.

```
void  fetch(word *instr_addr)
{
	enpipe((_instruction) *instr_addr);
}
```

The *decode()* stage is a bit more involved. In this stage, Crisp's instruction decode logic parses the instruction and generates necessary control signals that are needed for the 'execute' stage. The simulator can afford, however, to abstract most of these low-level details and only implement NEXTPC modification logic.

```
void  decode(_intruction *instr, _decoder_output *out)
{
	if (instr) {

		out->opc = get_opcode(instr);

		/* most Crisp instructions are in 3-address code format */
		get_operands(instr, &out->opd1, &out->opd2, &out->opd3);
```

```
	} else {

		/* invalidate output so that execute stage ignores it */
		out->result = INVALID;
	}

	/* prepare REG_NEXTPC for next cycle's fetch stage */
	set_reg_val(REG_NEXTPC, get_reg_val(REG_NEXTPC) + 4);
}
```

In the *execute()* stage, Crisp's functional units such as the ALU, shifter and data memory interfaces are activated according to the control signals generated by the decode stage for this instruction in the previous cycle. The simulator only needs the decoded instruction for this phase.

```
void  execute(_decoder_output *decoded_instr)
{
	extern void (* instr_handlers[])(_operand1 *, _operand2 *,
_operand3 *);

	instr_handlers[decoded_instr->opc]
	(
		&decoded_instr->opd1,
		&decoded_instr->opd2,
		&decoded_instr->opd3,
	);

	/* prepare REG_PC for the next cycle's execute stage */
	set_reg_val(REG_PC, get_reg_val(REG_PC) + 4);
}
```

Each type of instruction is executed by its handler which can be obtained by indexing into *instr_handlers[]* with the instruction's opcode.

Sample handlers

The assembler instruction "*add r0, r1*" triggers the following handler in its execute stage:

instr_handlers[OPC_ADD](REG_R0, REG_R0, REG_R1);

This causes *opc_add_handler()* to be invoked with the three operands.

```
void  opc_add_handler(_operand1 *opd1, _operand2 *opd2, _operand3
*opd3)
```

```
{
	/*
	 * operand1 = operand2 + operand3;
	 * operand3 can either be a register or an immediate value;
	 */

	set_reg_val (
		opd1->reg,
		get_reg_val(opd2->reg) +
		(opd3->type == OPD_TYPE_REG) ?
			get_reg_val(opd3->reg) :
			opd3->imm
	);
}
```

A store instruction such as "*store r0, [r1], #4*" can be handled as follows:

```
void  opc_load_handler(_operand1 *opd1, _operand2 *opd2, _operand3
*opd3)
{
	/*
	 * operand1 is the source register;
	 * operand2 is the base register containing the memory address;
	 * operand3 can either be a register or an immediate value and
	 *          specifies an offset from the base;
	 */

	set_mem_val (
		get_reg_val(opd2->reg) +
		(opd3->type == OPD_TYPE_REG) ?
			get_reg_val(opd3->reg) :
			opd3->imm,
		get_reg_val(opd1->reg)
	);
}
```

Handling branches

Arithmetic and logical instructions such as *add/sub, shift, or/xor/and* and *compare* update an internal register - REG_FLAGS, which holds processor state information related to carry, overflow, zero etc. Program flow can be altered by branching based on the state of these flags. This helps implementation of control structures such as *if-else*, *for* and *do-while* using branch instructions.

Branch instructions break the smooth flow in the pipeline and hence need special handling. *Crisp* takes the non-interlocked approach in implementing branches, by unconditionally executing two instructions that immediately follow the branch instruction in the program. This is also known as *delayed-branching* and the two

instructions following the branch instruction are said to be in *branch-delay-slots*[9]. This approach helps the pipeline to run without stalling for the branch target to be fetched.

```
void  opc_branch_handler(_operand1 *opd1, _operand2 *opd2, _operand3
*opd3)
{
      extern int  pending_bds;      /* to handle branch delay slots */

      /* operand1 can be one of:
       *  a register containing the target address;
       *  a PC-relative target address offset as a +/- immediate value;
       *
       * operands 2 and 3 are not applicable to branch instructions
       */

      set_reg_val (
            REG_NEXTPC,
            (opd1->type == OPD_TYPE_REG) ?
                  get_reg_val(opd1->reg) :
                  get_reg_val(REG_PC) + opd1->imm
      );

      pending_bds = 2;  /* next two cycles are branch delay slots */
}
```

The number of branch delay slots yet to be executed is tracked by *pending_bds*. This helps to maintain the integrity of REG_PC. The REG_PC updation logic in *execute()* has to be modified to handle branches. During the execution of the two delay slots REG_PC should contain their addresses but should contain the address of the branch target immediately after the completion of the delay slots' execution. This is accomplished by resetting REG_PC to trail REG_NEXTPC by two instructions, as should be the normal case.

```
void  execute(_decoder_output *decoded_instr)
{
      extern void (* instr_handlers[])(_operand1 *, _operand2 *,
_operand3 *);

      extern int  pending_bds, bds_flag;
                  /* to handle branch delay slots */

      instr_handlers[decoded_instr->opc]
      (
            &decoded_instr->opd1,
            &decoded_instr->opd2,
            &decoded_instr->opd3,
```

[9] It is possible to do with one delay-slot but pipeline behaviour illustration is easier by allowing two of them

```
	);

	/* prepare REG_PC for the next cycle's execute stage */
	switch (pending_bds) {
		case 2:
			/*
			 * first of the two delay slots executed in this
			 * cycle; let REG_PC move forward
			 */
			set_reg_val(REG_PC, get_reg_val(REG_PC) + 4);
			pending_bds --;
			break;

		case 1:
			/*
			 * second delay slot executed in this cycle;
			 * reset REG_PC to trail REG_NEXTPC
			 */
			set_reg_val(REG_PC, get_reg_val(REG_NEXTPC) – 8);
			pending_bds --;
			break;

		case 0:	/* normal sequential flow */
			set_reg_val(REG_PC, get_reg_val(REG_PC) + 4);
	}
}
```

To avoid indeterminate behaviour, the *Crisp* architecture suggests that a BDS may not contain a branch instruction[10].

REG_PC and the pipeline

Readers would have noticed that special control logic is required to ensure that the user-visible REG_PC always contains the address of the current instruction being executed. However, if REG_PC were allowed to reflect the state of the pipeline, this control logic can be eliminated. For instance, in the ARM processor architecture, the PC value is two instructions ahead of the current instruction being executed. The ARM programmer has to factor this while performing any calculations based on the PC value.

This space left intentionally blank

[10] Readers are welcome to experiment with code sequences that violate this suggestion, to understand the motivation.

A sample run

We shall now take a simple *Crisp* assembly program ‘sigma_10’ and run it on the simulator we designed in the previous section.

```
        ; sigma_10: a program to compute the sum of first 10 natural
        numbers

              .text
        sigma_10_start:

0x00    MOV   r0, #10     ; number count
0x04    MOV   r1, #0      ; current sum

        sum_loop_start:
0x08          ADD   r1, r1, r0  ; r1 := r1 + r0
0x0c          SUB   r0, r0, #1  ; r0 := r0 – 1
0x10          CMP   r0, #0      ; are all the numbers done ?
0x14          BNE   sum_loop_start
0x18          NOP               ; BDS-1: no operation
0x1c          NOP               ; BDS-2: no operation

        ; at exit, r1 contains 55, the required sum.
              HALT              ; halt the processor

        sigma_10_end:
```

Shown below are 11 clock cycles of *Crisp* along with the processor state and the actions taken in each of the pipeline stages. Of special interest are cycles 7 through 10, which illustrate REG_NEXTPC/REG_PC behaviour when branches are encountered in the pipeline.

Cycle #0:
***Crisp* state:**
NEXTPC: 0x00
PC: -0x08
pending_bds: 0

Actions:
Fetch(0x00) i.e., *MOV-1*
Decode(-), NEXTPC += 4
Execute(-), PC += 4

Cycle #1:
***Crisp* state:**
NEXTPC: 0x04
PC: -0x04
pending_bds: 0

Actions:
Fetch(0x04) i.e., *MOV-2*
Decode(MOV-1), NEXTPC += 4
Execute(-), PC += 4

Cycle #2:
***Crisp* state:**
NEXTPC: 0x08
PC: 0x00
pending_bds: 0

Actions:
Fetch(0x08) i.e., *ADD*
Decode(MOV-2), NEXTPC += 4
Execute(MOV-1), PC += 4

Cycle #3:
***Crisp* state:**
NEXTPC: 0x0c
PC: 0x04
pending_bds: 0

Actions:

Fetch(0x0c) i.e., *SUB*
Decode(ADD), NEXTPC += 4
Execute(MOV-2), PC += 4

Cycle #4:
***Crisp* state:**

NEXTPC: 0x10
PC: 0x08
pending_bds: 0

Actions:

Fetch(0x10) i.e., *CMP*
Decode(SUB), NEXTPC += 4
Execute(ADD), PC += 4

Cycle #5:
***Crisp* state:**

NEXTPC: 0x14
PC: 0x0c
pending_bds: 0

Actions:

Fetch(0x14) i.e., *BNE*
Decode(CMP), NEXTPC += 4
Execute(SUB), PC += 4

Cycle #6:
***Crisp* state:**

NEXTPC: 0x18
PC: 0x10
pending_bds: 0

Actions:

Fetch(0x18) i.e., *NOP-1*
Decode(BNE), NEXTPC += 4
Execute(CMP), PC += 4

Cycle #7:
***Crisp* state:**

NEXTPC: 0x1c
PC: 0x14
pending_bds: 0

Actions:

Fetch(0x1c) i.e., *NOP-2*
Decode(NOP-1), NEXTPC += 4

Execute(BNE), NEXTPC = 0x08, PC += 4, pending_bds = 2

Cycle #8:
***Crisp* state:**

NEXTPC: 0x08
PC: 0x18
pending_bds: 2

Actions:

Fetch(0x08) i.e., *ADD*
Decode(NOP-2), NEXTPC += 4

Execute(NOP-1), PC += 4, pending_bds--

Cycle #9:
***Crisp* state:**

NEXTPC: 0x0c
PC: 0x1c
pending_bds: 1

Actions:

Fetch(0x0c) i.e., *SUB*
Decode(ADD), NEXTPC += 4

Execute(NOP-2), PC = NEXTPC - 8, pending_bds--

Cycle #10:
***Crisp* state:**

NEXTPC: 0x10
PC: 0x08
pending_bds: 0

Actions:

Fetch(0x10) i.e., *CMP*
Decode(SUB), NEXTPC += 4
Execute(ADD), PC += 4

Advanced design notes

Before calling it a day, allow me to add a few notes on the finer aspects of simulator design.

- In reality, the *Crisp* assertion that all instructions complete in 3-cycles is impractical. Allowance has to be made for memory latencies, load/store delays, multiplier output delays and the like. A memory-interface module can abstract the details of the memory heirarchy, associated buffers and latencies. This calls for altering the pipeline behaviour according to the processor specification.
- Interaction with co-processors (FPU/MMU etc) has not been covered in this paper. Individual processors can be designed as separate processes and inter-process communication facilities offered by the host OS can be used to communicate data and control signals between the processors. This makes the simulator modular and easy to implement.
- Interrupts and exceptions such as data aborts can be handled by using *setjmp* (for setting up exception handling code) and *longjump* (for handling an exception). User defined signals can also be used for this purpose.
- Speculative branching can be implemented by pre-fetching the target based on the probability of the branch being taken. The probability can be computed by maintaining a history of 'branch taken/not-taken' per branch instruction in the program.
- Some processors execute independent instructions out-of-order to improve throughput. The instruction stream can be converted into a dependency graph of code-blocks and then be executed out-of-order based on the dependencies. A thorough understanding of the target processor's instruction retiring policy is important to implement this feature. This feature can be abstracted off the simulator if appropriate allowance can be made to the resulting reduction in performance of the target processor being simulated.
- Fine-grain profiling can be performed by accessing the system-wide clock counter via appropriate interfaces (e.g., get_clock_ticks() and set_clock_ticks()) at required points of execution.

It is important however, to understand the requirements of the users before adding complex features to the simulator. In the absence of a demonstrated need (current/future) for modeling specific processor features / functional units, it is better to abstract them and keep the simulator simple and functional. For, the aim of a simulator is not to replace an FPGA prototype.

Conclusion

In this paper, we have seen the architectural differences that make a RISC processor simpler to design and yet extract better performance as compared to a CISC processor. Software modeling of processor behaviour was explained by designing a simulator for a hypothetical RISC processor – *Crisp*. A sample run of the simulator demonstrated the inner workings of the *Crisp* pipeline. Design tips for simulating advanced techniques employed in modern RISC processors were only briefly discussed towards the end so as to keep this paper focussed at embedded systems designers and programmers who are new to the RISC philosophy. Readers can gain further knowledge about RISC concepts from the references.

References

- David A. Patterson and David R. Ditzel, *The case for the reduced instruction set computer,* Computer Architecture News, October 1980.

- David A. Patterson and John L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface,* Morgan Kaufmann Publishers, 1997 (second edition), ISBN 1-55860-329-8
- Dave Jagger (editor), *ARM Architecture Reference Manual,* Prentice Hall
- Steve B. Furber, *ARM System Architecture,* Addison-Wesley, 1996, ISBN 0-201-40352-8
- *http://www.sun.com/microelectronics/sparc/* - Sun SPARC architecture related information on the web
- *http://www.research.ibm.com/journal/rd38-5.html* - IBM Journal of Research and Development*, POWER2 and PowerPC architecture,* Vol. 38, No. 5, 1994, Order No. G322-0194
- *http://www.arm.com/* - data sheets, instruction set summaries, white papers, application notes and architecture references for the ARM family of processors