CIS 629: Project No.3

RISC Q and A

Sameer S. Shende

December 6, 1994

1. To describe Microprogramming, we have to look at the organisation of the CPU. A processor consists of hardware registers, ALU ,three types of data busses (A,B and C) for operands to be fetched and written back into the registers. To execute an instruction, data needs to be put out on the appropriate data busses, it needs to be latched, the ALU needs to get the operands and the instructions, and executed, and again a significant amount of control signals need to be generated to synchronise the operations of the various units in the CPU. A Microinstruction consists of a series of bits (22 bits as described for a specific machine in the Tanenbaum text). A microinstruction thus controls the data path and the control path in a CPU affecting the Memory address register, memory buffer register, ALU operation, initiating a read or a write request ,selecting busses for data transfer between units etc.

   A microprogram consists of microinstructions and defines the instruction set. A microprogram is stored in a writable control store (area within the CPU for storing the equivalent of a big instruction set). A particular instruction can be implemented in the microprogram by using an interpreter for the microinstruction. The advantage of coding part of the software in microprogram was that new efficient (read complex) instructions could be tailor made for a specific software requirement and implemented for the processor as a set of instructions which resided in a control section of the processor (ROM or RAM). This storage area which controlled the operation of the processor (by means of the microinstructions it contained) was known as the "writable control store". So, a language similar to the assembly language was developed in which the microprogram was coded and interpreted to form micro-instructions which controlled the overall CPU operation. This is described as microprogramming.

2. A large number of instructions in the ALU are 3 operand instructions. These are Add, Subtract, Multiply, Divide on integers, floating point numbers etc. Typically, if we consider the add instruction A= B + C it needs two memory addresses to be generated to fetch the operands B and C (and load these in registers), then calculate the sum (ADD) and store the result in a register. This can be stored in a memory location subsequently. In 3 - address instructions, the intermediate stage of fetching the operands and then executing it is not done. There can be register-register (as described above), register-memory and memory - memory models of execution of instructions. In a memory - memory instruction, the address of the memory location where the address of the operand is located is specified and the address where the result has to be stored using the same indirection is specified. The three address instruction format which requires data to be fetched or written to memory three times in order to execute the instruction has the advantage that its very easy to implement and the compilers are easy to write for such a model. The obvious disadvantage for such a scheme is that it is inefficient. Consider the case when an address is generated from the CPU - this being the virtual address, it has an overhead of translation into the physical address, searching for it in a cache memory and if there's a cache miss fetching it from the main memory during which time the processor waits (in the absence of instruction pipelining).

   A RISC architecture uses instruction pipelines to concurrently fetch data and instruction required for a later instruction, and execute the current instruction. It also relies very heavily on the fact that it can reuse the data in the registers rather than getting it from memory. The data is not flushed out of the registers after an instruction is executed but retained for future use. RISC uses explicit LOAD and STORE instructions to get data. This way the 3 addresses need not be generated for every instruction. It relies on the compiler to optimally use the data in the registers. RISC tries to execute every instruction in just one clock cycle (at the minimum).

3. A delayed branch is used in RISC machines to minimise the bubbles appearing in a pipeline. The stages in the execution of an instruction is IF (instruction fetch which gets the instruction using the program counter), ID (instruction decode to find what the instruction should do), OF (operand fetch - which gets the operands needed for the instruction), OE (operand execution - the actual execution of the instruction) and OS (operand store - which stores the results). Since each memory reference takes a cycle multiple of these stages are pipelined where the operand execution stage can execute concurrently with the operand fetch stage of the next instruction). So, it is

critical that the pipeline is always full and all units are executing to give the maximum throughput.

However, when a branch instruction is encountered, the address of the next instruction cannot be determined till the actual execution of the branch which can cause a buuble to appear in the pipeline. A delayed branch executes the instruction following the branch instruction when the instruction fetch of the next instruction is in progress. So, the effect of the branch is delayed by one or more cycles. Thus, the pipeline is kept full and the bubble is not introduced in the instruction pipeline. This poses a difficulty to a the compiler writer who has to implement the delayed branch in the compiler so that the results of the execution of the instruction following the branch can be used. For e.g., Consider the execution of the following instructions

Address	**Instructions**	**With Delayed Branch**
1000	INC C	INC C
1001	SUB B,A	JUMP 1004
1002	JUMP 1004	SUB B,A
1003	ADD A,C	ADD A,C
1004	INC A	INC A

In this case, if the instructions were executed as shown in the first column, The jump instruction would have caused a bubble to appear. Instead if it is relocated as shown in the second column and the effect of the Jump is visible only after the SUB B,A has been done, then there's no bubble in the pipeline. Thus, instead of executing SUB and then JUMP, the JUMP is initiated and the operands for the 1004 are fetched at the same time as the execution of the SUB instruction. Thus, the optimised delayed branch can work just as well but introduce fewer bubbles in the pipeline. (It is important to note that in this case, the delayed brach worked because the destination of the branch or address did not depend on the executionof SUB instruction, if this was not the case, then the bubble would have appeared).

A delayed load is similar to the delayed branch instruction. A load or a store operation requires that the address be computed in one cycle and the data be fetched from the memory in the second cycle and used in the third cycle. During the execution of one phase the other phase of the next load can be active. So, in the case of a delayed load, the compiler puts another instruction in the pipeline while this is executing. From the paper by Patterson on RISC, they claim that it can be done 90machine.

4. RISC systems depend more heavily upon Compiler technology than the CISCs because the advantage that can be achieved by using the

data from the registers, and filling the pipeline optimally has to be addressed by the compiler. The optimum reuse of an item in a register (assuming a large number of registers) is done by the compiler. The compiler also checks if an operand generated by an instruction is required in the next cycle (independence of input-output) by the next instruction and if that is the case, then it reschedules the instructions so that there is no clash. This logic has to be built into the compiler in the case of RISC but is avoided in the CISC case. The compiler also takes care of keeping the pipeline full by using branch prediction, delayed branch and keeping track of branch history. By using virtual registers in the RISC architecture, if there's a conflict about using a register, then it can be sometimes resolved by using different physical registers for the same logical register. This can be also done by the compiler (but is currently implemented in hardware).

The compiler relies very heavily on registers and resolves all conflicts arising out of using the same register by successive instructions in a manner which may result in inconsistencies or create a bubble in the pipeline.

Thus, RISC architects rely on efficient compilers to extract high throughput from the processor.

5. Register windows are used by the RISC processor to ensure that arguments in a procedure or a function call are not copied again and the register that stores an argument can be used in the next context after the program counter is changed following the execution of the function call. If a certain set of registers is being used by a thread, and it calls a procedure, then the set of registers allocated to the called procedure could overlap with those of the calling procedure. This overlap is what can be termed as a "register window". Essentially, the underlying philosophy of the RISC architecture is to have as few loads and stores of operands as possible and to reuse these in the registers. So, one way to maximise reuse is to implement some optimising compilers to do this or to do it in hardware. While the possible opportunities for this reuse can be easily seen in the compiler implementation, these would be difficult to detect if we take the hardware approach. One area where a clear cut reuse pattern can be seen is argument passing in the function call implementation. So, in the absence of compiler experts in the Berkeley project, they chose to implement the register windows.

6. The clock cycles on some of the early RISC processors was longer than the corresponding clock cycle on CISC. For e.g., a T805 (from Inmos) came with a 20MHz clock while a i386 clock was 25Mhz, 33 MHz. However, the T805 had an onchip co-processor (FPU) and a 4KB cache (reduced instruction gave more silicon space) and so it

did outperform the Intel 386 even with a i387. However, the RISC clock frequency started increasing the CISC frequency and now it is definitely more than the CISC processors. For e.g., a i486 and a Sparc processor had 33 MHz and 50 MHz frequencies and Sparc II outperformed the i486. With the clock doubler chips (the DX2, DX4 chips) Intels 486 and Pentium perform at rates of 66, 75, 100 MHz and correspondingly the DEC AXP Alpha processor performs at 270 MHz (and will touch 400 MHz by next year). Also, PA-RISC 7100 LC processor performs at 99 MHz. So, we can conclude that the RISC processors today not only outperform CISC due to their pipeline design (e.g., MIPS R8000 @75MHz outperforms Pentium @100 MHz due to the floating point pipeline) but are having higher frequencies of operation as well.

7. The number of instructions in a RISC processor is very small and the instructions are simple. This gives a clear picture of the processor architecture to the compiler and gives the barest minimum constructs for the compiler to use. It also means that complex operatoions need to be implemented in software by the compiler but all the resources that the processor can offer in terms of the pipeline, registers, on chip cache, floating point hardware unit are all in control of the compiler and it can efficiently use these.

8. To see the MIPS R8000 there is a slight evidence of backsliding from RISC to the more complicated instruction set as seen in its Multiply and Add instruction (floating point a x + b) and because of the 5 stage pipeline that it operates on we can see 4 floating point operations per clock cycle (in all 20 in 5 cycles) which accounts for 75Mhz * 4 = 300 MFlops. Also, a Linpack performance of 200Mflops (1000 x 1000) could be achieved only because of this *complex* floating point instruction. In terms of compiler technology we see that in Power-Challenge series it not only does all the functions of a RISC compiler but there's a parallelising compiler as well which can create concurrent threads of execution on IRIX 6.0 64 bit O.S. from a sequential code. So, the trend seems to be to go for a more "hybrid" view of these technologies (Pentium has a dual stage pipeline - a feature thet it probably picked up from the RISC world, and in R8000 we see new exotic instructions which were the forte of the CISC world) and to pick out the best features of both the worlds.