

COMP3320/COMP6464: High Performance Microprocessors

Alistair Rendell

See: *High Performance Computing*, Dowd and Severance, Chapter 2
Structured Computer Organization, Tanenbaum, Chapter 8 in Ed 3
Great Microprocessors of the Past and Present, <http://www3.sk.sympatico.ca/jbayko/cpu.html>

5 Instruction Set Architectures

- Early microprocessors were very simple, but in 1964 IBM introduced the 360 series which was microprogrammed.
- From then instruction sets and addressing modes increased, prompted in part by development of high level languages.
- Special microcode was added to handle case statements, procedure calling, array indexing etc.
 - led to the CISC concept (Complex Instruction Set Computer)
- In the 70s writing, debugging and maintaining microcode became a major issue.
- Academics begin to analyse what programs actually did and this resulted in a major rethink of microprocessor design
 - led to the RISC concept (Reduced Instruction Set Computer)

5.1 Typical Program Instructions (as of late 1970s)

| Statement | SAL | XPL | Fortran | C | Pascal | Average |
|------------|-----|-----|---------|----|--------|---------|
| Assignment | 47 | 55 | 51 | 38 | 45 | 47 |
| If | 17 | 17 | 10 | 43 | 29 | 23 |
| Call | 25 | 17 | 5 | 12 | 15 | 15 |
| Loop | 6 | 5 | 9 | 3 | 5 | 6 |
| Goto | 0 | 1 | 9 | 3 | 0 | 3 |
| Other | 5 | 5 | 16 | 1 | 6 | 7 |

- **Conclusion:** most programs consist of assignments, if statements and procedure calls

5.2 Typical Assignments and Procedures (as of late 1970s)

| Assignment | | Procedures | |
|------------|----|------------|--------------|
| N Terms | | N Locals | N Parameters |
| 0 | - | 0 | 22 |
| 1 | 80 | 1 | 17 |
| 2 | 15 | 2 | 20 |
| 3 | 3 | 3 | 14 |
| 4 | 2 | 4 | 8 |
| ≥5 | 0 | ≥ 5 | 20 |

- 80% of all assignment are *variable = value*
- 21% of all procedures have no local variables
- 41% of all procedures have no parameters

Theoretically people can write highly complex programs, but the ones that they do write are actually very simple consisting of assignments, if statements and procedure calls with limited numbers of arguments

5.3 Comparison of CISC and Early RISC Machines

| | CISC | | | RISC | | |
|--------------------------|-------------------------------|-------------------------------|-----------------|------------|-------------------|------------------|
| | IBM 370/168 | VAX 11/780 | Xerox Dorado | IBM 801 | Berkeley RISC1 | Stanford MIPS |
| Year Completed | 1973 | 1978 | 1978 | 1980 | 1981 | 1983 |
| Instructions | 208 | 303 | 270 | 120 | 39 | 55 |
| Microcode size (bytes) | 54K | 61K | 17K | 0 | 0 | 0 |
| Instruction size (bytes) | 2-6 | 2-57 | 1-3 | 4 | 4 | 4 |
| Execution Model | Reg-reg Reg-mem mem-mem | Reg-reg Reg-mem mem-mem | Stack | Reg-reg | Reg-reg | Reg-reg |

5.4 Characteristics of RISC and CISC Machines

| | RISC | CISC |
|---|---------------------------------------|---|
| 1 | Simple instructions taking 1 cycle | Complex instructions taking multiple cycles |
| 2 | Only LOADS/STORES reference memory | Any instruction may reference memory |
| 3 | Highly pipelined | Not pipelined or less pipelined |
| 4 | Instructions executed by the hardware | Instructions interpreted by the microcode |
| 5 | Fixed format instructions | Variable format instructions |
| 6 | Few instructions and modes | Many instructions and modes |
| 7 | Complexity is in the compiler | Complexity is in the microprogram |
| 8 | Multiple register sets | Single register set |

Assembly language programmers used the complicated machine instructions, but compilers generally did not. Difficult to get compiler to recognise complicated instructions. Optimising compiler work by simplifying and eliminating redundant computations. After a pass through optimizing compiler opportunities to use the complicated instructions tend to disappear.

RISC is now the dominate scalar processor architecture

5.5 RISC Processors

First Generation Characteristics

- Pipelining (both instruction and floating point)
- Branching (delayed branching and branch prediction)
- Uniform instruction length
- Load/Store architecture (simple addressing)

Second Generation

- Faster clocks
- Superpipelining
- Superscalar

Post-RISC

- Out-of-order execution

5.6 Pipelining

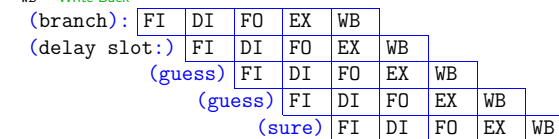
Everything happens in step with the clock. Reducing the clock time is good, but greater benefit comes from partially overlapping instructions so that more than one can be in progress at any time. RISC architectures can initiate an instruction each cycle, but previous instructions may not have completed.

- Break instr'n execution into k stages; \Rightarrow can get $\leq k$ -way ||ism

(generally, the circuitry for each stage is independent)

- eg. ($k = 5$): stages FI = Fetch Instrn., DI = Decode Instrn., FO = Fetch Operand, EX = Execute Instrn.,

WB = Write Back



- note: FO & WB stages may involve memory accesses (and may possibly stall the pipeline)

Pipelining: Dependent Instructions

CPU must ensure result is the same as if no pipelining (or ||ism)

- Instr'n's requiring only 1 cycle in the EX stage:

```
add %1, -1, %1      ! r1 = r1 - 1 (integer register subtract)
cmp %1, 0           ! is r1 = 0? (integer register compare)
```

can be solved by pipeline feedback on EX stage (to EX stage, next cycle)

- (important) Instr'n's requiring $c > 1$ cycles for the EX stage (ie. f.p. *, +, load, store) are normally implemented by having c EX stages. This requires the dept. instr'n to be delayed by 3 cycles eg. $c = 3$

```
fmuld %f0, %f2, %f4 ! I0: fr4 = fr0 * fr2 (f.p. register multiply)
....                ! I1:
....                ! I2:
faddd %f4, %f6, %f6 ! I3: fr6 = fr4 + fr6 (f.p. register add)
```

| | | | | | | | | | | |
|-----|----|----|----|-----|-----|-----|-----|-----|-----|----|
| I0: | FI | DI | FO | EX1 | EX2 | EX3 | WB | | | |
| I1: | | FI | DI | FO | EX1 | EX2 | EX3 | WB | | |
| I2: | | | FI | DI | FO | EX1 | EX2 | EX3 | WB | |
| I3: | | | | FI | DI | FO | EX1 | EX2 | EX3 | WB |

Pipelining: Dependent Instructions(cont)

Notes:

- If I3 is $\delta < c$ cycles after I0, the CPU must insert $c - \delta$ pipeline bubbles (NoOps) in between.
Can avoid this by software pipelining: (where possible) separate I3 from I0 in the original code by at least c cycles
- EX2, EX3 may be 'empty' for the simpler instructions (eg. int +)
- Less important instrn.'s requiring larger c (eg. f.p. /, int *, /, %) are either not pipelined or use a separate sub-pipeline for their EX stages

Pipelining: Branch Instructions

Branch to a new program address (perhaps caused by an if statement) will disrupt the pipeline flow. Processor doesn't know if instruction is a branch until the decode stage and then may not know if it will be taken until the execute stage. If branch is taken then following "in flight" instructions must be annulled.

- Many processors require a 'branch delay slot' instr'n immediately after the branch instruction. This enables the pipeline to continue for unconditional branches (ie when the decoded instructions says branch somewhere and we go there). Conditional branches are more difficult, pipeline will stall and require flushing as it can't be recognized before the DI stage
eg.

```
cmp %1, %2          ! n = n + 1
bne endif1         ! if (i == k) ...
add %3,1,%3        ! delay slot - ALWAYS executed
```

(if possible try to move a logically preceding instr'n into the delay slot)

Pipelining: Branch Prediction

- To handle conditional branches various branch prediction schemes are used:
 1. Assume branches are *always* taken (flush pipeline when not taken)
(OK for loops, with test at bottom)
eg. SPARC, SuperSPARC
 2. S/W (compiler) indicates the '*most likely*' prediction (UltraSPARC)
 3. H/W keeps a branch prediction buffer: predict using result of the *last* (few) executions of the branch (2 bits, on UltraSPARC)

5.7 Pipelines and Floating-Point Operations: Summary

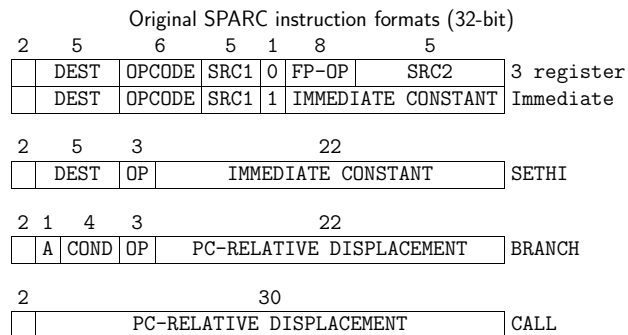
- FP operations typically take longer than fixed point operations so benefit greatly from pipelining.
 - Number of stages in the pipeline may be increased so even complicated operations like FP * can be pipelined.
- FP +, -, *, comparison and conversion pipelined
 - Usually sqrt and / are NOT pipelined
- Some processors limit overlap of FP operations due to shared internal components (e.g. for normalization of number)
 - Fully pipelined ⇒ no overlap restrictions

5.9 Load/Store Architecture

- Memory reference restricted to load/store.
 - Only one reference per instruction.
 - In CISC arithmetic/logical instruction may include memory reference.
- Motivation:
 - To enable fixed instruction length
 - To ease pipelining
 - Since memory reference may be slow

5.8 Uniform Instruction Length

- Fundamental RISC feature.
- Eases decomposition into pipeline stages.
- CISC machine had no apriori knowledge of length of each instruction.



5.10 Second Generation RISC Processors

After proving basic concept

- Improvement in manufacturing led to faster clock rates (esp. DEC)
- Increase pipeline stages making each stage simpler and faster (eg MIPS)
- Add multiple compute elements: SuperScalar

5.11 SuperScalar (multiple instr'n issue)

A small number (w) of instr'ns are scheduled by the H/W to *execute together*

- Groups must have an appropriate 'instruction mix'
 - eg. UltraSPARC ($w = 4$): $\left. \begin{array}{l} \leq 2 \text{ different floating point} \\ \leq 1 \text{ load / store ; } \leq 1 \text{ branch} \\ \leq 2 \text{ integer / logical} \end{array} \right\} \text{instr'ns per group}$
- Have $\leq w$ -way ||ism over *different types of instr'ns*
- Generally requires:
 - Multiple ($\geq w$) instr'n fetches
 - Extra grouping (G) stage in the pipeline
- **Problem:** will require a deeper software pipelining (by a factor of w)
 - Generally, all problems with pipelining are similarly amplified
- **Issues:** the instruction mix must be balanced for maximum performance!
 - NB. floating point *, + must be balanced in any case

5.12 Post-RISC Architecture

- Two-way superscalar successful and in 1994 able to run at 1.6-1.8 instructions per cycle.
- "Higher-way" superscalar may appear natural progression, but difficult to find instruction level parallelism to justify.
- Speculative execution or out-of-order execution is more popular. Permits instructions to be executed that may never be used, e.g. in the following FDIV may be elevated up the execution stack if sufficient space is present to store the result.

```
LD    R10,R2(r0)    Load into R10 from memory
.
.
.
.
.
FDIV  R4,R5,R6      R4 = R5 /R6
```

- Out-of-order processors include a *instruction reorder buffer* to store instructions that are in limbo.

5.13 Summary

- RISC is now the dominant architecture type.
 - Pentium is a mix of a CISC and RISC processor
- Typical pipelines are 5-15 stages and instructions are 3-4 way superscalar.
- Can only achieve up to inherent parallelism in instruction stream.
- **Dependent instr'ns** must be *sufficiently* separated by either:
 1. S/W (need good compilers & large # registers)
 2. H/W (if done via *dynamic instr'n reordering*, this is more effective, but harder to achieve!)