10. Pipelined, RISC- and Super Scalar Processors

(a) Pipelining.
(b) RISC and CISC machines.
(c) Super scalar processors and instruction-level parallelism.
Laundry Analogy: Assume several people want to carry out laundry. We have the following 4 stages:
1. Wash the clothing using the washing machine.
2. Dry it in a dryer.
3. Fold clothing.
4. Put them away.
This can now be done in a pipeline:
- When you have washed your clothing, the next one can use the washing machine.
- Similarly for the dryer, table for folding clothing.
The following slide illustrates the speed up.
Picture omitted for copyright reasons
Pipelining of the Instruction Cycle

The Instruction Cycle can now be divided into similar pieces.
Assume a sequence of instructions of the form:
- LOAD R1, (R1 + A)
  
  i.e. the second argument has addressing mode base-register displacement.
We divide it into 5 stages:
  
  1. Instruction fetch (1 cycle).
  2. Register load (1/2 cycle).
  3. ALU operation (1 cycle) (for calculating R1 + A).
  4. Data access (1 cycle).
  5. Register save (1/2 cycle).
All 5 parts can be carried out independently.
(For the increase of the PC by one in 1. one needs an extra adder). The next slide demonstrates the speed up.
It is ideally by the factor 4.
Remark

We don’t increase the speed of a single operation, but the throughput of instructions. (In the laundry example, everybody needs the same amount of time as before for his laundry).

Problems

- The stages require different amount of time. The pipeline is only as fast as the slowest device. Can be improved, but not avoided by using as small stages as possible.
Problems (Cont.)

- **Data dependencies** between the stages. Eg. the second instruction requires a register, in which a value was stored by the first register:
  
  **Example:** First instruction increments register R1 by 1. Second instruction loads data register indirectly from memory using register R1.

  **Solutions:**
  - Either second instruction has to be delayed till first instruction is ready.
  - Or (nowadays standard) **pass on the data from one unit to the other** if possible.

- **Main problem** are **branches**, since we do not know, whether the next instruction is the one at the next location or we have to follow the branch.

The next slide illustrates the additional delay, we get, if we have to wait until the condition of a branch is computed.
Picture omitted for copyright reasons
Solutions for Problem with Branches

- **Delay next instruction** till decision is taken. Takes much time (branch instructions occur frequently).

- **Always continue both with next instruction and follow the branch.**
  If it is determined which was the next instruction, throw away the other one.
  **Problem:** might require unnecessary data access with removal of data from cache.

- **Branch Prediction:**
  Guess the next instruction. If the guess was wrong, throw away the instruction (and don’t change memory or registers before this decision is taken).
Strategies for Branch Prediction

- **Branch never taken.** Always fetch the next instruction.
- **Branch always taken.** Always follow the branch (i.e., assume temporarily that the branching condition is true).
- **Predict by opcode** Decide by the kind of branch whether to follow the branch or not.
- **Taken/not taken switch.** Add to the instruction bits which reflect history taken. Usually stored when instruction is in cache or special buffer.
- **Branch history table.** Store information in this table. Information not lost if instruction removed from cache.
Loop buffer

Some sort of cache, but
- only for instructions,
- stores always a sequence of instructions.
Instructions are **prefetched** if possible so that even if a small **forward jump** is done, often the next instruction is already in the buffer. Loop buffers are **well suited for looping structures**, if loop fits within the buffer. When a branch is reached, the next instruction can be often be obtained from the loop buffer.

Delayed Branch

Used in RISC machines. One reorders the instructions so that the next instruction can always be used. If none can be found, insert an instruction **NOP** (no operation). Reordering to be done at compile time.
Conflict of Resources

In general there will be conflict of resources. Most resources can be doubled for this purposes.

**Exception:** Access for memory. Required for instruction fetch, data load and data storage.

**Solution:**
- Divide cache into instruction cache and data cache.
- Since usually instructions are not rewritten, unproblematic.
- Loop buffer serves often already as cache memory.
Example: MIPS

Next slide shows the preparation of a CPU for pipelining.

- **Two extra adders** were added.
  (for calculating the next instruction: one for addition of 1 to PC and one for addition of an offset to PC in case of a jump).
- Separate accesses to data and instructions build (corresponds to separate caches).

The slide afterwards shows the pipelined version, where extra registers (IF/ID, ID/EX, EX/MEM, MEM/WB) where added which store the data between the stages (step to and from them requires 1/2 memory cycle each).
Picture omitted for copyright reasons
Picture omitted for copyright reasons
The stages of the pipeline are
- Instruction fetch (in the area up to the first bar).
- Register fetch and instruction decoding.
- Arithmetic operation.
- Save result in main memory.

This is optimal only for instructions which apply an arithmetic operation to two register contents and store the result in main memory. For other operations the order of use of the stages of the architecture is altered and some stages are used more than once, which leads to a delay of the pipeline (next instruction has to wait until current one is completed).
(b) RISC and CISC Machines

(i) CISC Machines.

(ii) Analysis of CISC Machines.

(iii) RISC Machines.

(iv) CISC vs. RISC and Current State.
(i) CISC Machines

- CISC = complex instruction set computers.
- **Semantic Gap**: Gap between constructs in higher-level programming languages (HLL) and assembly languages instructions.
- **Problems** of increasing use higher-level program constructs:
  - Decreasing efficiency.
  - Increasing size of the compiled program.
  - Increasing complexity of compilers.
- Attempt to close it by introducing more and more complex instructions. This resulted in:
  - Increase of number of machine instructions and addressing modes.
  - Instructions with long execution time which require complicated hardware.
  - Long instruction formats in order to accommodate a lot of instructions.
(i) Analysis of CISC Machines

Studies carried out in the late 70’s and early 80’s on usage of machine instructions in computers.

Results:

- **Complex instructions** are not exploited by compilers.
  Possible reasons:
  - They are too specific.
  - Optimizing generated code easier with simple instructions.

- **Assignment statements dominate**, therefore simple movement of data is of high importance.

- **Conditional and unconditional branches** occur as well frequently.

- **Procedure calls and returns are most time-consuming operations** in HLL.

- Most references to variables are to **simple scalar variables** (integer, floating point or strings which store one item only as opposed to arrays, lists etc.).
• Most such references refer to **local variables**.
• Procedures have usually only **few parameters** (rarely more than 6) and only **few local scalar variables**.
• During execution for a long time one remains within a **rather narrow sequence of instructions** (window). This window moves only slowly over time.
(iii) RISC Machines

From the above the following conclusions were drawn.

- Instead of creating complex instructions one should optimize performance of most time-consuming features of HLL.
- The following main principles of RISC machines were suggested in order to achieve this:
  - Use of a large number of registers. Try to use them as much as possible. Effective since variables are often local and registers are accessed much faster.
  - Delay branches and delay loads to make pipelining easier and avoid the choice of the wrong branch.
Main principles of RISC machines, (Cont.)

- Reduce the number of instructions, instruction length and instruction format and fix instruction length and format. (See slide 260 for a RISC instruction set compared with slide 259 for a CISC instruction set with varying instruction length).

The reason for this is:

* Fixed instruction format and length makes pipelining more easier, more uniform and therefore faster.

* Simple instructions are faster executed and pipelined and the hardware can be more optimized.
Register Files and Register Windows

- Use of many registers in a register file, which is organized in a circular way. See slide No. 361.
- When a new procedure called the following happens:
  - Parameters which pass data from the calling procedure to the called one and back again are saved in new free registers of the register window. These parameters are temporary registers for the calling procedure and parameter registers for the called procedures.
  - Further new local registers are assigned for the local variables of the new register.
  - Parameter-, local and temporary registers of a procedure together form the register window of the procedure.
  - The register windows overlap.
Picture omitted for copyright reasons
• The total number of registers is usually fixed. (Typical size is 16 or 32 registers per window).
• The window allows only to accommodate a few most recent procedure activations (limited nesting of procedure calls). Typical numbers are 8, 16, 32.
• If a procedure call requires more registers than available, the content of old procedures are stored back into main memory.
• When returning from a procedure, the registers of which have been overwritten, the registers have to be loaded back from main memory.
• Two pointers needed for organizing this:
  – the current-window pointer (CWP) points to the end of the currently used register area in the window.
  – the saved window pointer (SWP) points to the beginning of the currently used register area in the window.
Because of the limited depth of recursions, the storage of registers into main memory is not often needed.

For global variables (common to several procedures) some additional global registers might be provided. On the next slide the circular register file is illustrated.

- A.in, B.in ... are the parameter registers of procedure A, B, ...
- A.loc, B.loc ... are the local variables of procedure A, B, ..., and
- w0, w1 ... are the register windows of A, B, ....
Picture omitted for copyright reasons
Comparison Cache vs. Register Files

- Register Windows and cache operate in a similar way.
- Cache can however as well store non-scalar variables, instructions etc.
- Advantage of registers is that they can be addressed with very few bits, and that the address computation is simple and fast.
Some RISC architectures have not as many registers. Then one instead optimizes compilers, so that they try to use registers in such a way that transfers to and from memory are minimized.
RISC Pipelining

- In RISC one allows always the next instruction to be executed even in case of a branch.
- In case of branches therefore instructions have to be reordered. For instance the branch can be moved forward, if it doesn't depend on the previous operation.
- In case the there is no suitable next instruction, a NOP (no operation) instruction is to be inserted.
- Similarly, in case of dependency of one instruction on a previous one because of register use, reordering of instructions is applied.
- RISC pipelines are therefore very efficient, since the pipeline is most of the time in full use.
RISC instruction sets

Characteristics of RISC instruction sets are

- **One instruction per machine cycle.**
  (Machine cycle = length it takes to fetch two operands from registers, perform an ALU operation and to store the result in a register).
  That results in a uniform pipeline.

- Most operations should be **register-to-register**, only load and store operations access memory.
  This simplifies the instruction set and therefore the control unit.

- Use of **simple addressing modes**, so that addresses can be computed fast.

- **Simple instruction formats** (only few different ones). Allows to do opcode decoding and register operand accessing simultaneously.
(iv) RISC vs. CISC and Current State

- Examples of full RISC processors are MIPS R4000, Sun SPARC, Power PC.
- Not clear whether RISC machine are so much faster.
  Examples of directly comparable RISC and CISC machines are missing.
- Tendency towards mixture of CISC and RISC features:
  - Addition of some CISC instructions to RISC architectures.
  - Especially Intel Pentium nowadays have an initial interpretation procedure, in which the fetched CISC instructions are interpreted as internal RISC instructions which are then evaluated in a RISC pipeline.
• Difficulty of Intel to move from the CISC instruction set (would destroy compatibility with previous processors).
• It seems to be easier to market sophisticated CISC instruction sets. (Compare with advertisement of cameras, stereo equipment etc. in which one points out the new switches they have, even so they are rarely used).
• Tendency towards integration of CISC and RISC.
(c) Super Scalar Processors and Instruction-Level Parallelism

(i) From super pipelining to super scalar.
(ii) Design issues for super scalar architectures.
(iii) Full instruction-level parallelism.
From Super Pipelining to Super Scalar

- **Super pipelined approach:**
  Use the fact that if data is to be passed from one register to the next one, it arrives half a clock cycle later instead of a full clock cycle later. Use pipeline stages so that every half clock cycle the next instruction is fetched. See next slide.

- **Super scalar approach:**
  - Execute several instructions fully in parallel.
  - Requires to have multiple functional units, all of which are implemented as a pipeline. For instance one has two integer ALUs, two floating point ALUs, but usually only one access to memory.
Picture omitted for copyright reasons
• Origin of name super scalar because super scalar processors are optimized for processing **scalar data**, as opposed to **vector processors**, which are optimized for processing **vector data** (arrays) in parallel. The latter are important for numerical applications (eg. weather forecast, other simulations of physical systems, engineering).

• Super scalar is now the technical standard for processors in computers (there are as well processors in cars, calculators ...). Power PC from RS/6000 onwards, all Pentium processors, MIPS R10000, Sun’s UltraSPARC-II are examples of it.

• Super scalar approach is an instance of **instruction-level parallelism**: Instruction level parallelism means that multiple instructions are executed in parallel.
Limitations of Super Scalar Architectures

Instructions can not always executed in parallel because of:

- **True data dependency** between multiple instructions (the next instructions depends on the result of a previous one).

Example:

\[ R1 := R1 + 3; \]
\[ R2 := R1 + 4. \]

Partial solution: pass result of one computation directly to the next instruction (in the example, as soon as the result \( R1 + 3 \) of the first instruction is obtained, it can be used in the second one at the same time as storing it in \( R1 \)).
• **Procedural dependency.** Dependencies on branching instructions. Can be solved as in pipelining by *branch prediction*. (*Delayed branch strategy* for RISC architectures turns out to be less effective).

• **Resource conflict.** (Two instructions require the same functional unit of the CPU). Two instruction may use the same resource. Can be resolved by adding more resources. However, access to memory and registers is limited.
- **Output dependency.** If two instructions store values in the same register, the later one has to be completed after the first one: E.g. in case of
  \[ R1 := R2 + R3; \]
  \[ R17 := 3; \]
  \[ R1 := 3, \]
  if the third instruction is completed before the first one, at the end a wrong result might be stored in \( R1 \).

Solution: **Register renaming.** Register numbers in the instructions are virtual, which are mapped by some logic to actual registers.

In the above example, the \( R1 \) in the third instruction (and in later instructions) is mapped to a different actual register than the \( R1 \) used in the first instruction.
• **Antidependency.** If one instruction reads a register which is overwritten by a later instruction, the later one shouldn’t be completed before the first one.

Example:

\[
R1 := R2 + 3;
R2 := 3.
\]

If \( R2 := 3 \) is completed before \( R2 \) is fetched in the first one, the first instruction has a wrong result.

Can be solved again by register renaming. **Register renaming** turns out to be crucial in order to obtain any substantial speed-up.
Design Issues for Super Scalar Processors

Usually instructions are not fully independent. Three orderings are important:

- Order in which instructions are fetched.
- Order in which instructions are executed.
- Order in which result of an instructions is stored in registers or memory.

The terminology used is

- **Issue** for the order in which fetched instructions are passed on to the units executing it.
- **Completion** for the order in which their results are stored in registers and main memory.
Three Policies for Super Scalar Architectures

- **In-order issue with in-order completion.**
  Both issue and completion of instructions in the order they occur in the program. If there is a conflict execution of later instructions is delayed. Rarely used.

- **In order issue with out-of-order completion.**
  Now completion can be out of order. However decoding of instructions only done up to the point of dependency or conflict. More complex instruction-issue logic required.
- **Out-of-order issue with out-of-order completion.**

  Addition of a new buffer, called **instruction window**.
  
  - When an instruction is decoded it is placed in the instruction window.
  - As long as this buffer is not full, new instructions will be decoded.
  - When a function unit in the execution stage is available, an instruction from the instruction window may be issued, provided no conflicts or dependencies block this instruction.

Therefore the processor has lookahead capability, ie. it can identify independent instructions ahead of the current one.
Super Scalar Execution

In the next slide an overview over super scalar execution of programs is shown.

- In the **instruction fetch process**, a dynamic stream of instructions is created. **Branch prediction logic** is applied simultaneously. Dependencies are examined and if possible removed.
- The instructions are then **dispatched** into a **window of execution**. This is done in the order of dependency instead of the order in the program.
- At the end instructions are **issued** to the **retiring unit**.
- The retiring unit guarantees that data conflicts when writing data to memory and registers are resolved, so that the result is as in an execution in true order. Results not yet retired are kept in temporary storage.
  If a wrong branch was taken, wrong results have to be thrown away.
Picture omitted for copyright reasons
Units Required in Super Scalar Architectures

- Good instruction fetch strategies combined with branch prediction logic.
- Logics for determining and if possible solving true dependencies.
- Mechanisms for issuing multiple instructions in parallel.
- Resources for parallel execution of instructions, both for functional units and for access to main memory (in parallel!).
- Mechanisms for designing the retiring unit.
(iii) Full Instruction-Level Parallelism

- Plan of Intel, together with HP: IA64, with code name for first processor Itanium.
- Original code name was Merced.
- Step towards 64 bit processors.
- As for super scalar processors
  - large number of registers (256 64-bit register and 64 1-bit predicate registers).
  - multiple execution units.
- Starts completely new family of processors with completely new instruction sets.
  (Break in an over 20 years long history, starting in the late 1970s with the x86 family).
New Ideas in the Itanium

- **Explicit parallelism.**
  New instruction format, which holds 3 instructions, which can be executed in parallel.
  Additional information in the instruction, which allows to execute several of these instruction bundles in parallel.

- Therefore resolving of dependencies and determining of order done at compile time rather than during execution.
  Compiler has much more time to resolve such issues.
New Ideas in the Itanium (Cont.)

- **Predicated Execution:**
  In order to allow the execution of many instructions in parallel, they are predicated (has to be done at compile time).
  If one instruction contains a condition (e.g. “if $x = 3$”), and other instructions depend on positive and negative outcome of this (e.g. the “then”- and “else” clause in an if-then-else construct), two predicates (from the predicate registers) are chosen.
  They will be assigned as soon as the condition is resolved, one becomes true if the condition is true, the other if it is false.
  The instructions which are to be executed depending on this condition will now be prefixed with these predicates.
Execution of all of them can be done now in parallel. However the prefixed instructions will only retire when the condition is known and the corresponding predicate is set to be true. If it is set to false they will be dismissed.

**Example of predication**

“If \(a = b\) then \(k := k + 1\) else \(l := l + 1\)” is translated into

\[
P2, P3 := \text{compare}(a, b)
\]

\[
< P2 > k := k + 1
\]

\[
< P3 > l := l + 1.
\]
New Ideas in the Itanium (Cont.)

- **Speculative load.**
  - Bottle neck is access to main memory. Therefore we want to move load instructions ahead, so that they can be carried out as soon as access to main memory is possible.
  - Problem with moving load instructions over branch instructions: A wrong load might raise an exception, since it refers to an invalid memory address. (Some memory boundary might be crossed, but this exception will not occur when executing the program in correct order, because the instruction is not to be executed).
  - Solution: Introduction of new instructions, which replace an ordinary load instruction:
* Speculative load (written for instance as “ld.s. R4, (R3)”). Loads main memory content into a register, but doesn’t deliver the result, and doesn’t raise an exception.

* Checking instruction “chk.s”, at the place where the original load instruction was. In the example above, “chk.s R4” would express: deliver the result of a previous speculative loading of R4 to R4, and raise an exception, if an exception occurred.

If there is a branch in between, chk.s might be predicated, which means that the result of the loading is only delivered and an exception is only raised in case the predicate is true.
Current Development

- Explicit parallelism seems to be the right track.
- Problems with marketing, since in order to make use of the full power of the Itanium, operating systems and compilers have to be completely rewritten (step to a completely new instruction set), and software needs to be adapted.
- Degree of parallelism will most likely increase in the near future.
- It seems that Linux support will be available earlier than versions of Windows for the Itanium.