Computer Architecture: summaries

December 22, 2003

1 Instruction Set Architectures

Name: Arend van Beelen
Studentno.: 0220507
LIACS email: dvbeelen@liacs.nl

This summary is mainly about Instruction Set Architectures, or ISA in short, but first I'll cover a few fundamentals of Computer Architecture.

1.1 Computer Architecture Fundamentals

The items covered are *not* a complete coverage of Computer Architecture Fundamentals, just a few topics that were covered during the second class of the course.

1.1.1 Amdahl's Law

One of the first important fundamentals of CA is Amdahl's Law. Amdal's Law states that any performance enhancement is limited to the time that is being spent in the part in which the enhancement has been made.

We can represent this as:
$Speedup = \frac{Performance\ for\ entire\ task\ using\ the\ enhancement\ when\ possible}{Performance\ for\ entire\ task\ without\ using\ the\ enhancement}$

1.1.2 General abbreviations and formula's

A very important abbreviation is MIPS, which stands for "millions of intstructions per second". Note that the MIPS value is dependant on the instruction set architecture (more about that later) and that a higher MIPS value does not necessarily means better performance.

Another abbreviation is CPI, or clocks per instruction. The CPI value, combined with the clock frequency of a CPU, can be used to count the MIPS value of a CPU:
$MIPS = \frac{Clock\ rate\ (MHz)}{CPI}$

The CPI value itself can be calculated using the weighted average. For this you take the clocks cycles of an instruction and multiply it with the time you spend in that particular instruction. Do this for every instruction and take the average of that.

The CPU time that's needed to execute a certain can be calculated by multiplying the instruction count (IC) of the task by the CPI and the time needed for one clock cycle. The time needed for one clock cycle on a 500 MHz CPU is 2ns.

1.2 Instruction Set Architectures

An instruction set architecture (ISA) is defined by its instructions, its addressing modes, types and operands. An instruction is an elementary CPU instruction. Address modes are register, immediate, displacement, register indirect, direct/absolute, etc...

An ISA can be seen as the layer between the assembly language and the CPU. It determines how the CPU can be programmed from the assembly language. In order:

High Level Language $->$ Assembly $->$ ISA $->$ CPU

1.2.1 ISA Classes

There are basically four classes of ISA's. Namely Stack, Accumulator and two classes of Register computers. A Stack architecture has no registers, all its operations are done on the top of its stack. An Accumulator has one implicit register, the accumulator. It performs its operations on the accumulator and the memory. The Register computers have multiple explicit registers which they can use for multiple purposes. The first type of Register computer is the Register-Memory architecture which use both its memory and the registers in its operations. The second type, called Load-Store or Register-Register architecture can only use the main memory in explicit load/store operations. Almost any modern computer uses Load-Store Register architecture, including RISC processors.

Instructions are loaded from memory (pointed to by the program counter (PC) to the PCPU. The instruction is executed. PC is increased.

1.2.2 Little Endian/Big Endian

Values are represented in binary either using the Little or the Big Endian scheme. With Little Endian the least significant byte (LSB) is located at the most left byte. Big Endian (which looks more like our written representation of numbers) is the opposite and places the LSB at the most right byte. *Please note that the Endianness has no influance on the bit-order.*

It is important to note that numbers sometimes have to be converted from Little Endian to Big Endian or the other way around. For example, Intel processors use Little Endian where most network protocols and mainframes use Big Endian. When data is exchanged between conflicting Endians, the byte-order needs to be reverted.

1.2.3 Instruction alignment

Words (4 bytes) are placed in memory according to a given alignment. This means that if a word is placed at address A, and s is the size of the word, then $Amods == 0$. Therefore, every word may only begin on every 4th byte.

1.2.4 Instruction encodings

Instructions can be encoded in several ways, namely:

Variable encoding Operation plus a variable number of operands.

Fixed encoding Operation plus a fixed number of operands.

Hybrid encoding Some operations have a variable number of operands, some operations have a fixed number of operands.

The biggest advantage of fixed encodings is the ease of decoding, while variable encodings are generally more efficient regarding memory usage.

1.2.5 ISA Design

- number of explicit operands
- operand storage
- type and size
- operations

Stack and accumulator have implicit operands.

1.3 RISC

RISC is the most popular ISA nowadays, and we will now look at the RISC architecture in particular. One of its primary advantages is its simplicity which allows for efficient compilers. RISC processors have only three types of operations: arithmetic, memory and control flow.

1.3.1 Arithmetic operations (ALU)

Arithmetic instructions are adding, substracting, bit-shifting and the like. Examples:

```
ADD R1, R2, R3    # adds R3 to R2 and stores the result in R1
ADDI R1, R2, 10   # adds the value 10 to R2 and stores the result in R1
ADDU ...          # same as ADD but for unsigned values
ADDUI ...         # same as ADDUI but for unsigned values
```

Bitwise shift operations simply move all bits of a register to the left or the right. The new bits are filled with zeros. If you want to do a shift to the right operation which keeps the sign bit, you should use arithmetic shifting, as opposed to logical shifting.

1.3.2 Memory operations

Memory operations allow you to load and store registers from/to memory. Some commands:

```
LB   # load byte
LBU  # load unsigned byte
LH   # load half-word
LHU  # load unsigned half-word
SB   # store byte
SBU  # store unsigned byte
etc...
```

If you want to address some memory relative to an address stored in a register you can use a displacements. For example, the displacement `100(R2)` means to take the address stored in R2 and add 100 bytes to it.

1.3.3 Control flow operations

Control flow operations are operations that manipulate the program counter (PC). They are used to create if-then-else statements, create loops and procedure calls.

Branches are conditional jumps. They jump to a label if they evaluate to true and continue if false. Example:

```
BEQZ R1, name   # branch equal zero
```

The previous statement jumps to `name` if R1 equals zero and does nothing if otherwise.

Unconditional jumps simply jump to a label (eg. `J name`) or to an address specified by a register (eg. `JR R1`).

Finally, there's a special jump used for procedure calls. It jumps to a label or an address stored in a register and saves the current program counter to R31. You can then call the procedure 'foo' by executing `JAL foo` and return from the function using `JR R31`.

2 Pipelining

Name: Wouter Meuleman
Studentno.: 0234826
LIACS email: wmeulema@liacs.nl

This summary deals with the subject of pipelining, as discussed in Appendix A of the third edition of 'Computer Architecture, A Quantitative Approach', by Hennessy and Patterson, hereafter refered to as 'H&P'. The concept and performance of pipelining are discussed, aswell as possible hazards that might occur while using it.

2.1 Concept of pipelining

Pipelining is an implementation technique whereby multiple instructions are overlapped in execution. This basically means that while one instruction is in a certain phase of the instruction cycle (not the first), the next instruction is already going through an earlier phase of the cycle. A big advantage of this technique is the fact that less resources remain idle, having a faster throughput as a result in most situations.

A good analogy for the concept if pipelining is a DIY laundry service. A laundry cycle could consist of the following phases:

1. washing (takes 90 minutes)
2. drying (takes 120 minutes)
3. ironing (takes 60 minutes)

This means a total laundry cycle takes $(90 + 120 + 60)$ 270 minutes. Now imagine three persons all wanting to do their laundry. Also imagine this particular DIY laundry service to be a rather small one, containing no more than one washing machine, dryer and ironing board. If each consecutive person politely waits for the previous person to finish doing his or her laundry, the total time needed for all four persons to finish would be at least (3×270) 810 minutes.

However, this process can be sped up. After the first person has finished washing and has moved his or her laundry into the dryer, the washing machine comes available again. At this point the second person can already start using it. This process may repeat itself for the other person and other phases of the laundry cycle. This is the concept of pipelining. Figure A.1 in H&P gives a clear example of a simple pipeline consisting of 5 phases and 5 instructions or 'pipes' (persons wanting to do their laundry).

2.2 Performance

While we mentioned a faster throughput when using pipelining, this does not mean pipelining improves the latency of a single task; each phase of the laundry cycle still takes the same amount of time as it did before using pipelining. However, pipelining does improve the throughput of the entire workload (i.e. three persons doing their laundry).

The speedup of a pipelined system compared to its original non-pipelined version can be computed using the following equations:

$$Speedup = \frac{Average\ instruction\ time\ on\ original\ system}{Average\ instruction\ time\ on\ pipelined\ system} \quad (1)$$

$$= \frac{CPI_{original}}{CPI_{pipelined}} \times \frac{Clock_{original}}{Clock_{pipelined}} \quad (2)$$

In the ideal case we hope to obtain a $CPI_{pipelined}$ that is consistent with the following equation, where the depth of the pipeline indicates the number of phases in the pipeline:

$$CPI_{pipelined} = \frac{CPI_{original}}{Depth\ of\ pipeline} \quad (3)$$

However, in practice this kind of $CPI_{pipelined}$ will hardly ever be reached. This is due to the fact that a pipelined system almost always needs to cope with a certain amount of overhead.

Another reason Equation 3 usually does not hold is that the slowest phase in an instruction cycle determines the speed of all other phases (*The strength of a chain is only determined by its weakest link*). To clarify this, we will again use the laundry service analogy. After the first person has finished washing and starts drying, the second person starts washing. However, as soon as the second person finishes washing, he or she can not start drying. This is because the drying phase takes 120 minutes, so the first person is going to be using the dryer for 30 more minutes, for which the second person has to wait.

Now that we have briefly discussed the performance of pipelined systems, we can calculate what the speedup of a pipelined DIY laundry service would be. In order to simplify things, we will now assume that the DIY laundry service venue is already crowded with people doing their laundry and that there will not come an end to this. In other words, the pipeline is already completely filled and will continue to be so. This means that after *each* phase transition, one laundry cycle will be completed. Furthermore, we know that each phase in the pipelined laundry cycle takes 120 minutes (i.e. the time it takes for the longest phase to complete). With this information and Equation 1 we can calculate the speedup:

$$Speedup = \frac{90 + 120 + 60}{120} = 2.25$$

2.3 Implementation

Section A.3 of H&P explains in some detail how to implement a pipelined system for a MIPS architecture. Because the text found in that section is already quite compact, it will not be compacted any further in this summary. We will however mention an aspect of the implementation of a pipelined system that is quite fundamental; the addition of extra registers between instruction cycle phases.

In order to ensure that instructions in different phases of the pipeline do not interfere with eachother, 'pipeline registers' have been introduced. These are registers that exist between phases of the pipeline; they actually connect the two phases together. At the end of a phase, all results of that phase are stored in the pipeline register. In the next phase, these results can be used as input, but they can also be passed on to the next pipeline register between the now-current phase and the next.

Although pipeline registers are a good mechanism for avoiding errors in a pipelined system, there are still a number of unwanted situations that might occur.

2.4 Hazards

Situations can occur that prevent the next instruction in the instruction cycle to be executed during its designated clock cycles. These situations are called ‘hazards’. We distinct three types of hazards.

2.4.1 Structural hazards

Structural hazards arise when different instructions want to make use of a specific piece of hardware at the same time in such a way that the hardware can not execute all instructions simultaneously. In other words, when a resource conflict arises.

Structural hazards can be avoided by adding more hardware or by improving the instruction scheduler. If neither of these solutions is an option, the pipeline will have to be ‘stalled’ temporarily. This is done by using a so called ‘pipeline bubble’ or just ‘bubble’. Using this bubble, one of the instructions in conflict will be stalled for one clock cycle so that the other instruction can use the resource. Ofcourse, as a result of this, all instructions after the stalled instruction will also be stalled for one clock cycle. Figure A.5 in H&P shows a clear example of an instruction that is stalled because of a structural hazard.

2.4.2 Data hazards

Data hazards arise when an instruction depends on the results of a previous instruction in such a way that it can not be accessed when needed. Figure A.6 in H&P shows an excellent example of this; before one instruction can write a result into a register, three later instructions are already trying to use that result.

Data hazards can be avoided by a technique having multiple, ofcourse equally impressive, names: ‘forwarding’, ‘bypassing’ and ‘short-circuiting’. We will use the term ‘forwarding’ from now on. The result that the first instruction in Figure A.6 tries to write back, is actually already known two phases before, after the ALU operation. Because the next instruction needs this result as an input to its own ALU operation, we can forward the result from the pipeline register after the ALU operation of the first instruction to the input of the ALU operation of the second instruction. Figure A.7 in H&P illustrates this.

Unfortunately, forwarding is not always possible in case of a data hazard. If the second instruction tries to use a result of the first instruction that is not yet known by the first instruction before the current phase, the pipeline will have to be stalled. This situation can occur when, for instance, the first instruction loads data from memory and the second instruction needs this data during the same clock cycle for an ALU operation. Because the phase where data is read from memory is right after the ALU phase, the memory phase of the first instruction and the ALU phase of the second instruction align in the pipeline. This is illustrated in Figure A.9 in H&P.

2.4.3 Control hazards

Control hazards are sometimes also called ‘Branch hazards’. When a branch is executed, it may or may not change the Program Counter (PC) to something other than the usual $PC + 4$. If a branch is *taken*, the PC is changed to its target address. However, this is not done before the Memory access (MEM) phase. This means that there might be a number of stalls before the next instruction can be fetched.

There exist a number of methods to avoid this situation as much as possible. One method is to move the Conditional check of the branch to an earlier phase, for example the Instruction Decode (ID) phase. This way, less stalls are needed.

Another method is to treat every branch as if it is not going to be *taken*, simply allowing the hardware to continue with the next instruction on the next pipe. One has to be sure not to write back anything to the processor, because these results might have to be discarded if later turns out the branch has to be taken.

One could also do the opposite, and assume that the branch will be taken. In this case, the next instruction at the branch target is fetched and executed. Again, nothing may be written back, because the branch might not have to be taken after all. Normally this method is not very helpful, as we usually get to know the target address and the branch outcome at the same time.

A fourth method is called 'delayed branch'. Using this method, the compiler recognises the branches in a program and decides what other instructions, that are independent of the branch, can be executed while waiting for the outcome of the branch condition. If used correctly, the behaviour of a delayed branch, and thus the performance, is the same whether or not the branch is taken. Figure A.14 in H&P shows the three ways in which delayed branching can be implemented, or more specifically, the three methods of selecting the correct independent instructions to be executed.

2.4.4 Impact of stalling on performance

Ofcourse, stalling has an impact on the performance of the pipeline. Assuming an ideal $CPI_{pipelined}$ of 1, we can say the following:

$$Speedup = \frac{CPI_{original}}{1 + Pipeline\ stall\ cycles\ per\ instruction}$$

In case all instructions take the same number of cycles, which also equals the depth of the pipeline, the following holds:

$$Speedup = \frac{Depth\ of\ pipeline}{1 + Pipeline\ stall\ cycles\ per\ instruction}$$

3 Pipelining: Basic and Intermediate Concepts

Name: Sjoerd Meijer
Studentno.: 0226467
LIACS email: smeijer@liacs.nl

3.1 Introduction

Pipelining is an implementation technique whereby multiple instuctions are overlapped in execution; it takes advantage of parallelism that exists among the actions needed to execute an instruction. In a pipeline different steps are completing different parts of different instructions in parallel. Each of these steps is called a *pipe stage* of a *pipe segment.* The *throughput* is determined by how often an instructions exits the pipeline.
The time required between moving an instruction one step down the pipeline is a *processor cycle.* Because all stages proceed at the same time, the length of a processor cycle is determined by the time required for the slowest pipe stage. If the stages are perfectly balanced:

$$time\ per\ instruction\ (pipelined\ machine) = \frac{Time\ per\ instruction\ on\ unpipelined\ machine}{Number\ of\ pipe\ stages}$$

To sumarize the preceding the following 'pipelining lessons' can be defined:

1. doesn't help latency single task;
2. pipelining limited by the slowest stage;
3. multiple tasks operating simultanously;
4. potential speed-up = number of pipe stage.

3.2 The Basics of a RISC Instruction Set

RISC architectures are characterized by a few key properties, which dramatically simplify their implementation:

- All operations on data appy to data in registers and typically change the entire register
- The only operations that affect memory are load and store operations.
- The instructions format are few in number with all instructions typically being one size.

Most RISC architectures, like MIPS, have three classes of instructions:

1. *ALU instructions* – These instructions take either two registers or a register and a sign-extended immediate, operate on them, and store the result into a third register.
2. *Load and store instructions* – These instructions take a register source, called the *base register*, and an immediate field, called the *offset*, as operands.
3. *Branches and jumps* – Branches are conditional transfers of control. There are usually two ways of specifying the branch condition in RISC architectures: with a set of condition bits or by a limited set of comparisons between a pair of register and a zero. MIPS uses the latter.

3.3 A Simple Implementation of a RISC Instruction Set

To understand how a RISC instruction set can be implemented in a pipelined fashion, we need to understand how its is implementated *without* pipelining. Every instruction in this RISC subset can be implemented in at most 5 clock cycles. The 5 clock cycles are as follows:

1. *Instruction fetch cycle* (IF):
   Send the program counter (PC) to memory and fetch the current instruction from memory. Update the PC to the next sequential PC by adding 4 to the PC

2. *Instruction decode/register fetch cycle* (ID):
   Decode the instruction and read the registers corresponding to register source specifiers from the register file.

3. *Execution/effective address cycle* (EX):
   The ALU operates on the operands prepared in the prior cycle, performing on one of three functions depending on the instruction type: memory reference, register-register, register-immediate.

4. *Memory access* (MEM):
   If the instruction is a load, memory does a read using the effective addres computed in the previous cycle. If it's a store, then the memory writes the data from the 2nd register read from the register file using the effective address.

5. *Write-back cycle* (WB):
   Write the result into the register file, whether it comes from the memory system or from the ALU.

3.4 The Classic Five-Stage Pipeline for a RISC Processor

We can pipeline the execution described above with almost no changes by simply starting a new instruction on each clock cycle (see Figure A.1 on page A-7). Each of the clock cycles from the previous section becomes a *pipe stage.*

With the introduction of pipelining we have to make sure that we do not want to perform two different operations with the same data path resource on the same clock cycle. We must also ensure that instructions in different stages of the pipeline do not interfere with one another. This separation is done by introducing *pipeline registers* between successive stages of the pipeline, so that at the end of a clock cycle all the results from a given stage are stored into a register that is used as the input to the next stage on the next clock cycle. Figure A.3 on page A-9 shows the pipeline drawn with these pipeline registers.

3.5 Basic Performance Issues in Pipelining

Pipelining increases the CPU instruction thoughput (but it does not reduce the execution time of an individual instruction):

$$Speedup\ from\ pipelining = \frac{Avg.\ instruction\ time\ unpipelined}{Avg.\ instruction\ time\ pipelined}$$

4 Pipeline Hazards

There are situations, called *hazards*, that prevent the next instruction in the instruction stream from executing during its designated clock cycle. There are three classes of hazards:

1. *Structural hazards* arise from resource conflicts when the hardware cannot support all possible combinations of instructions simultaneously in overlapped execution.
2. *Data hazards* arise when an instruction depends on the result of a previous instruction.
3. *Control hazerds* arise from the pipelining of branches and other instructions that change the PC.

Hazards in pipelines can make it necessary to *stall* the pipeline. Avoiding a hazard often requires that some instruction in the pipeline be allowed to proceed while other are delayed.

4.1 Performance of Pipelines with Stalls

A stall causes the pipeline performance to degrade from the ideal performance.

$$Speedup\ from\ pipelining\ =\ \frac{Avg.\ instruction\ time\ unpipelined}{Avg.\ instruction\ time\ pipelined}$$

$$Speedup\ from\ pipelining\ =\ \frac{CPI\ unpipelined * Clock\ cycle\ unpipelined}{CPI\ pipelined * Clock\ cycle\ pipelined}$$

$$CPI\ pipelined\ =\ CPI\ ideal\ +Pipeline\ stall\ clock\ cycles\ per\ instructions$$

$$Speedup = \frac{CPI\ unpipelined}{1+Pipeline\ stall\ cycles\ per\ instructions}$$

4.2 Structural Hazards

If some combination of instructions cannot be accommodated because of resource conflicts, the processor is said to have a *structural hazard.* When a hazard is encountered, the pipeline will stall one of the instructions until the required unit is available. A stall is commonly called a *pipeline bubble* or just *bubble.* Figure A.5 on page A-15 shows the stall by indicating the cycle when no action occurs and simply shifting instruction 3 to the right.

4.3 Data Hazards

Data hazards occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on a unpipelined processor. Consider the pipelined execution of these instructions:

```
ADD R1,R2,R3
SUB R4,R1,R5
AND R6,R1,R7
OR  R8,R1,R9
XOR R10,R1,R11
```

As shown in Figure A.6 on page A17, the ADD instruction writes the value of R1 in the WB pipe stage, but the SUB instruction reads the value during its ID stage. This problem is called a *data hazard.* There are three classes of data hazards: read after write (RAW), write after read (WAR), and write after write (WAW). Figure A.6 is an example of a ”read after write” data hazard.

Minimizing Data Hazard Stalls by Forwarding

The problem described above can be solved with a hardware technique called *forwarding.* Forwarding works as follows:

1. The ALU result from both the EX/MEM and MEM/WB pipeline registers is always fed back to the ALU inputs
2. If the forwarding hardware detects that the previous operation has written the register corresponding to a source for the current ALU operation, control logic selects the forwarded result as the ALU input rather than the value read from the register file.

Figure A.7 on page A-18 demonstrates this process of forwarding to avoid data hazards.

Data Hazards Requiring Stalls

Unfortunately, not all potential data hazards can be handled by bypassing. Figure A.9 shows that a load instruction has a delay or latency that cannot be eliminated by forwarding alone. Instead, we need to add hardware, called a *pipeline interlock*, to preserve the correct execution pattern. In general, a pipeline interlock detects a hazard and stalls the pipeline until the hazard is cleared.

4.4 Branch Hazards

Control hazards can cause greater performance loss for our MIPS pipeline than do data hazards. When a branch is executed, it may or may not change the PC to something other than its current value plus 4. Recall that if a branch changes the PC to its target address, it is a *taken* branch; if it is *not taken*, or *untaken.*

Reducing Pipeline Branch Penalties

There are many methods for dealing with the pipeline stalls caused by branch delay; we discuss four compile time schemes in this subsection.

1. The simplest scheme to handle branches is to *freeze* or *flush* the pipeline, holding or deleting any instruction after the branch until the branch destination is known.
2. A higher-performance scheme is to treat every branch as not taken, simply allowing hardware to continue as if the branch were not executed.
3. An alternative scheme is to treat every branch as taken. As soon as the branch is decoded and the target address is computed, we assume the branch to be taken and begin fetching and executing the target.
4. The fourth scheme in use in some processors is called *delayed branch.* In a delayed branch, the execution cycle with a branch delay of one is

```
branch instruction
sequential successor1
branch target if taken
```

The sequential successor is in the *branch delay slot.* This instruction is executed whether or not the branch is taken. The pipeline behavior of the five-stage pipeline with a branch delay is shown in Figure A.13 on page A-24. To conclude this subsection, Figure A.14 illustrade 3 different ways, being ”from before”, ”from target” and ”from fall-through”, of how the branch delay slot can be scheduled.

4.5 How is Pipelining Implemented?

After having discussed the basic concepts of pipelining in the previous sections, this section (at least in ”Hennessy & Patterson”) reviews a simple implementation of an unpipelined and a pipelined version of MIPS. I think it is worth to have a look at the following figures:

- Figure A.18 on page A-31, the data path is pipelined by adding a set of registers.
- Figure A.19 on page A-32, events on every pipe stage of the MIPS pipeline.
- Figure A.20 on page A-34, situations that the pipeline hazard detection hardware can see by comparing the destination and sources of adjacent instructions.

In my opinion, mentioning the Figures A.18-A.21 in this summary suffice. The figures illustrate theory discussed earlier in this summary. It is better to read these examples from the book.

Second Part of Pipeling

Name: Tsoe Loong Li
Student no: 0223948
Liacs email: tli@liacs.nl

5 A4: What Makes Pipelining Hard to implement

Exceptional situations are harder to handle in a pipelined CPU because the overlapping of instructions makes it difficult to know wheter an instruction can safely change the state of the CPU. (For the overview of the types exceptions, see page A-39 and the tabel on page A-40):
The requirements on exceptions can be characterized on five semi independent axes: (for details see book page A-41):
1. Synchronous vs asynchronous
2. user requested vs coerced
3. user maskable vs nonmaskable
4. within vs between instructions
5. resume vs terminate
If a machine can handle an exception, save it current state and restart then the machine is restartable.
The most difficult exceptions have two properties:
1. they occur within the exception
2. they must be restartable
Whenan exception occurs, the pipeline control can take the followin steps to save the pipelining state safely:
1. force a trap instructions into the pipeline on the next IF
2. until a trap is taken, turn of all writes for the faulting instructions and for all instructions that follow in the pipeline 3. after the exception handling routine in the OS receivers control, it immediately saves the PC of the faulting instructions.
A pipeline is said to have precise exceptions of the pipeline can be stopped so that:
instructions just before the faulting instructions are completed
instructions just after it can be restarted from scratch
For other exceptions such as floating point exceptions, the faulting instruction on some processors writes its result before the exceptions can be handled. In such cases the hardware must be prepared to retrieve the source operands, even if the destination is identical to one of the source operands.
Because floating-point operations may run for many cycles, some other instruction may have written the source operands. To overcome this, many recent high-performance CPU's have introduced two modes:
1. imprecise exceptions
2. precise exceptions
exceptions may occur out of order; that is, an instruction may cause an exception before an earlier instruction cause one.
Solutions is to post all exceptions caused by a given instruction in a states vector associated with that instruction. The exception status vector is carried along. Once an exception indication is set, turn of all writes. When instructions enters WB, the exception status vector is checked and exceptions are handle in order.

6 A5:Extending the MIPS pipeline to handle multicycle operations

Latency : is the numbers of cycles it takes for the functional unit to complete its operation.

initiation interval: number of cycles you have to wait before the functional unit can be re-used.
WAW only happens if two instructions write to the same registers and there is no intervening read.
In A5 we are gonna learn how MIPS pipeline can be extended to handel floating-point operations.
Floating poin operations need several cycles to complete. Many operations can be pipelined themselves. We imagine the floating point instructions have the same pipeline as integer with the exception that:
EX may be repeated
there may be multiple functional units
We assume there is four separate units:
-main integer unit
-FP and integer multiplier
-FP adder
-FP and integer divides. (see figure A.29 of book page A-48)
Figure A.30 show the latencies and inition intervals for functional units. The example pipeline structure in figure A.30 allows up to four outstanding FP adds, seven outstanding FP/integer multiplies and one FP divide.
Figure A.31 show how this pipeline can be drawn by extending figure A.29 (page A-49).
Hazards and Forwarding in longer latency pipelines
1. div unit not fully pipelined
2. instructions varying running times
3. instructions no longer reach WB in order, WAW hazard possible
4. instructions may complete in different order
5. because longer latency of operations

WAW only happens if 2 instructions write to the same registers and there is no intervening read.

Assume that the pipeline does all hazard detection in ID, there are three checks that must be performed before an instruction can issue.
1. check for structured hazards
2. check for RAW data hazard
3. check for a WAW data hazard.

In the book a example about how to deal with precise exceptions is explained. I will not explain in detail anymore. The solution
-ignore problem and except imprecise exceptions
-buffer result until all previous operations have completed
-keep PCs of all instructions that have not yet completed and exceptions rountine emulates the completion of these instructions
-only issue an instruction if it is quaranteed that all previous instructions will complete without exceptions

7 A6: Puttin it all together: MIPS R4000 pipeline

In this section, the MIPS R4000 pipeline is explained and superpipelining technique.
-several pipe stages can be executed fast:
Register Fetch, ALU operation
-decrease cycle time and use several stages for IFetch, memory access
-but branch penalties will increase
For detail see book page A60.

The 4 major causes of pipeline stalls or losses:
1. load stalls: delays causing from the use of a load result 1 or 2 cycles after load 2. branch stalls: 2 cycle stall in every taken branch plan unfilled or canceled branch delay slots
3. FP result stalls: stalls because of RAW hazards for an FP operand
4. FP structural stalls: delays because of issue restrictions arising from conflicts for functional units in hte FP pipeline

8 A7: Another view: The MIPS R4300 pipeline

I don’t think this section is very relevant.

9 A8: Crosscutting issues

For the basis structure of a MIPS processor with a scoreboard, see figure A51 in book.
The four steps which replace the ID, EX and WB steps are as follows:
1. issue: if unit free and no other active instruction has some dest register, instruction to unit
2. Read operands: monitor availability of operands (= no active instr. issued earlier will wirte it.) If available tell unit to proceed reading and begin execution.
3. execution: when result ready, notify scoreboard
4. write resultL check for WAR hazards and staff if necessary

Because operands are only read when both operands available in the registerfile, no advantage is taken of forwarding.
There are three parts to the scoreboard:
1. instruction status: indicates which of the four steps the instruction is in 2. functional and unit status: indicates the state of the functional unit. There are 9 fields for each functional unit:
busy
OP-operations to perform in the unit
FI-destinations register
Fj, Fk- source registers
Qj, Qk- functional units producing Fj, Fk
Rj, Rk- flag: Fj, Fk ready and not read? (set to no when read)
3. Register result status- indicates which FU will write each register (or blank if none).

10 Reducing average memory access time (AMAT)

Name: Joris Huizer
Studentno.: 0239534
LIACS email: `jhuizer@liacs.nl`

The performance of a memory hiarchy should be calculated as
`average memory access time:`

Average memory access time = Hit time + Miss rate $\times$Miss penalty

- `Hit time`
  is the time to hit in the cache
- `Miss rate`
  is the fraction not found in the cache
- `Miss penalty`
  is the time to replace a block from lower level, including time to replace in CPU

Therefor, the focus is not just on the Miss rate!
The classical approach to improve cache behavior is to reduce miss rates. There are three categories of misses:

1. Compulsory:
   The first access to a block is allways a miss
2. Capacity:
   The cache cannot contain all blocks needed during execution of a program
3. Conflict:
   Blocks may be discarded and later retrieved again, if too many blocks map to it (for example, in a direct mapped cache)

There are five techniques to improve cache behavior by reducing miss rates:

1. Larger block size:
   Increasing the block size will reduce compulsory misses (spacial locality) but the miss penalty is increased (the number of blocks in the cache is reduced) This may in fact increase the miss rate.
2. Higher associativity:
   `2:1 cache rule of thumb:`
   A direct-mapped cache of size N has about the same miss rate as a two way set-associative cache of size N/2. This is true for cache sizes less than 128 KB. Greater associativity can come at the cost of increased hit time.
3. Larget caches:
   Just increase the capacity of the cache. Problems are a longer hit time and a higher cost. However, this is done in off-chip caches.

4. Way Prediction and pseudoassociative caches:
   In way prediction, extra bits are kept in the cache to predict the block of the next cache access. This means the multiplexor is set early to select the desired block - so only one tag comparison is performed. A miss results in checking the other blocks for matches in following cycles.

5. Compiler Optimizations:

   - Reordening procedures in memory, reordening instructions (McFarling, 1989)
   - loop interchanging: Making a loop walk through values in cache if possible, instead of the order the programmer wrote it
   - blocking: Try to reduce the number of misses via improved temporal locality. By repeatedly accessing small blocks instead of just walking through the hole column/row of an array

As the AMAT (average memory access time) shows, inprovements in miss penalty can be as beneficial as miss rate improvements. As the memory gap grows, the relative cost of miss penalties increases.
The following optimizations address improving on miss penalty

1. Multilevel cache:
   Adding another level of cache allows the first level cache to be small, so it matches the clock cycle time of the CPU, and most access go to the second level cache so the mis penalty is reduced. (The AMAT formula needs to be refined so the miss penalty of the first cache includes the second level cache properties) Because the second level cache is not tied to the clock cycle time, it can use more advanced designes than the 1st level cache. The size should be much bigger.

2. Critical Word First and Early Restart
   These are the two strategies to minimize the amount of time the CPU needs to wait for the word:

   (a) Critical wordt first:
   Request the missed wordt first from memory and send it to the CPU as soon as it arrives; While the CPU continues, fill the rest of thee wods in the block

   (b) Early restart:
   Fetch the words in normal order, but as soon as the requested word of the block arrives, send it to the CPU, letting the CPU continue it's work

   The benefit is low unless cache blocks are large. Because of spatial locality, the CPU might request another word which is going to the same cache block yet, but hasn't arrived yet.

3. Giving Priority to Read Misses over Writes:
   This optimization serves reads before writes have been completed.

   - With a write-through cache the most important improvement is a write buffer. On a read miss the write buffer must be checked before reading the memory
   - With a write-back cache, the cost of writes can also be reduced; When a read miss will replace a dirty memory block, the dirty block is copied to a buffer. then memory is read, and finaly memory would be written. This way, the CPU won't have to wait untill the write has finished.

4. Merging write buffer:
   As both write-through and write-back cahces use a write buffer, there is allways some optimization there. For each new block to write, the block plus the full address is placed in the buffer. When the buffer allready contains the address, the new block is combined with the other block; this is called write merging. This optimization uses the memory more efficiently.

5. Victim Caches:
   The approach is to remember what was discared, in case it is needed again. This is done using a small, fully associative cache between a cache and its refill path. The blocks are checked on a miss before going to the next lower-level memory.

These are techniques to overlap the execution of instruction with activity in the memory hiarchy.

1. Nonblocking Caches to Reduce Stalls on Cache Misses:
   For pipelined computers that allow out-of-order completion, the CPU need not stall on a cache miss. A `nonblocking cache` or `lockup-free cache` allows the data cache to continue to supplu cache hits during a miss. This ”hit under mis” optimization reduces the effective miss penalty. It might allow for overlapping multiple misses and thus lower the miss penelty even further. However, this increases the complexity of the cache controller.

2. Hardware Prefetching of Instructions and Data:
   Prefetching is frequently done in hardware outside of the cache. The processor may fetch two blocks on a miss: the requested block and the block after that one. The second block is then put in a buffer.
   Data can also be prefetched. This can increase the data hit rate significantly.

3. Compiler-Controlled Prefetching:
   There are two flavors of compiler prefetching:

   (a) Register prefetch: loading the value into a register

   (b) Cache prefetch: loading data only into the cache

   Eiter of these can result in a page fault; if they do, they are simply turned into a no-op. Prefetching only makes sence if the processor can proceed while the prefetched data are being fetched; The data cache for such computers is normally unblocking.

11 Computer Architecture: Improving Cache Performance

Pieter Jordaan

Virtual Memory

A program could be so big that it's bigger than the amount of RAM available. There are 2 solutions for this problem.

- Overlaying: a procedure is saved to disc if it's not needed
- Virtual Memory

Overlaying is old fashioned, so we don't discuss overlaying.

Timesharing

Virtual memory is especially used for timesharing. Timesharing is a technique to let multiple applications run at the same time by different users. This is done with the "context switch" process which changes the running application after a certain number of milliseconds.
This means every program has less memory available and could also read memory from other programs. Virtual Memory prevents this from happening. Virtual memory gives every application a small part of the memory, but it let the program "think" it can use all the memory. If a memory address is non-existent or not available the value is retrieved from the hard disc. This makes it also possible to have multiple applications using the same memory address without having any conflicts.

Virtual Memory Manager

The virtual memory manager wil change a virtual memory address in a physical address and will retrieve the value from the hard disc if the value of the virtual memory address is not available(a page fault) in the memory. This is called memory mapping or address translation.
Every application has a base register pointing to the page table in memory. This page table tells what is in the virtual memory location, etc.
Virtual Memory works just the same as cache. Instead of a cache miss you'll get an address/page fault. A page fault however could take 6 million clock cycles and is therefore pretty slow. Therefore the replacement policy is to replace the least recently used memory address. There is a usebit which tells you if the address is used. There's also a diry bit which tells if the value has to be rewritten to file or that the value is unchanged.
Page tables have the following properties:

- Each memory reference needs two memory accesses
- A Translation Look-aside Buffer(TLB) is used as small cache to speed up the translation by storing the most recent page tabel entry.
- Page tables are paged
- One memory reference can cause 2 page faults

- there are also protection bits to permit write and / or read to that memory location
- Each process has its own page table. To prevent processes accessing other processes there's a base and bound register in hardware where base $\leq$ memory address $\leq$ bound are the only allowed memory addresses.
- Only the Operating System can change these registers!

Traditional Caches still use the physical address. Only a virtual cache uses virutal addresses. Also a context switch makes the cache absolete. This can be prevented by throwing away the cache or use an process identifier tag in every cache line. By using a write-through cache you can solve this problem as well.

12 Instruction Level Parallelism, Dynamic Scheduling

Name: Bogumila Sobolewska
Studentno.: 0170232
LIACS email: bsobolew@liacs.nl

12.1 Introduction

Pipelining can overlap the execution of instructions when they are independent of one another. This potential overlap among instructions is called *Instruction Level Parallelism* ILP since the instructions can be evaluated in parallel.

- Amount of parallelism in basic block (a straight-line code without branches) is quite small
- Dynamic branch frequency in integer programs is 15-25 % (what means that branch is expected after every 4-7 instructions)
- These instructions are also likely to depend on each other
- Need ILP across multiple basic blocks
- Simplest way: parallelism among loop iterations (loop-level parallelism) example:
```
for (j=0; j<1000; j++)
x[j] = x[j] + y[j];
```

Techniques for converting such loop-level parallelism into ILP:

- unroll loops statically by compiler
- unroll loops dynamically by hardware

Goal of this Chapter - Make Superscalar Processors Work

12.2 Evolution

1. Detect hazards in hardware and enforce pipeline stalls (basic correctness)
2. Out of order completion
3. Scoreboard out of order execution
4. Superscalar multiple instruction issue using available functional units in parallel, hardware hazard check
5. Wider superscalar add more functional units Make CPI ¡ 1

IMPORTANT:
”The goal of both our software and hardware techniques is to exploit parallelism by preserving program order *only where it affects the outcome of the program*. Detecting and avoiding hazards ensures that necessary program order is preserved.”

Data Dependence	Potential Hardware Hazard
*True Dependence*b=a; c=b;	RAW read after write
*Anti-Dependence*b=a; a=c;	WAR write after read
*Output Dependence*b=a; b=c;	WAW write after write

12.3 Recall from Pipelining

Pipeline CPI = Ideal pipeline CPI + Structural Stalls + Data Hazard Stalls + Control Stalls

- **Ideal pipeline CPI:**measure of the maximum performance attainable by the implementation
- **Structural hazards:**HW cannot support this combination of instructions
- **Data hazards:**Instruction depends on result of prior instruction still in the pipeline
- **Control hazards:**Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps)

Two instructions that are dependent:

- are not parallel;
- cannot be reordered.

Consider code :
`Loop: L.D F0,0(R1)` ; F0 is array element
`ADD.D F4,F0,F2` ; add scalar in F2
`S.D F4,0(R1)` ; store result
`DADDUI R1,R1,#-8` ; decrement pointer by 8
`BNE R1,R2,Loop` ; branch if R1!=R2
dependences : ADD.D from L.D (F0), S.D from ADD.D (F4),
BNE from DADDUI (R1)
Instruction j is **data dependent** on instruction i iff

- instruction i produces result that may be used by j
- or instr j is data dependent on instruction k and instr k is data dependent on j.

Name dependency

- 2 instructions use same register or memory location (called name)
- but there is no flow of data between the instructions

Examples:

- *antidependence* - j writes a register/memory location that j reads and i is executed first
- *output dependence* - i and j write the same register/memory location, Ordering must be preserved.

Ideas to Reduce Stalls

Technique	Reduces
Dynamic scheduling	Data hazard stalls
Dynamic Branch prediction	Control stalls
Issuing multiple instructions per cycle	Ideal CPI
Speculation	Data and control stalls
Dynamic memory disambiguation	Data hazard stalls involving mem
Loop unrolling	Control hazard stalls
Basic compiler pipeline scheduling	Data hazard stalls
Compiler dependence analysis	Ideal CPI and data hazard stalls
Software pipelining and trace scheduling	Ideal CPI, data and control stalls

- Dependences are a property of programs
- Presence of dependence indicates potential for a hazard, but actual hazard and length of any stall is a property of the pipeline
- Importance of the data dependencies
    1. indicates the possibility of a hazard
    2. determines order in which results must be calculated
    3. sets an upper bound on how much parallelism can possibly be exploited
- Next, look at HW schemes to avoid hazard

  Program order: order instructions would execute in if executed sequentially 1 at a time as determined by original source program HW/SW goal: exploit parallelism by preserving appearance of program order
- modify order in manner that cannot be observed by program
- must not affect the outcome of the program
  Instructions involved in a name dependence can execute simultaneously if name used in instructions is changed so instructions do not conflict
- Register renaming resolves name dependence for regs
- Either by compiler or by HW
- `add r1, r2, r3`
- `sub r2, r4,r5` renaming those two reg. will solve the dependence
- `and r3, r2, 1`

Control Dependencies
Every instruction is control dependent on some set of branches, and, in general, these control dependencies must be preserved to preserve program order.

```
if p1 {
S1;
```

```
};
if p2 {
S2;
}
```
S1 is control dependent on p1, and S2 is control dependent on p2 but not on p1.
Control dependence need not always be preserved - willing to execute instructions that should not have been executed, thereby violating the control dependences, if can do so without affecting correctness of the program
Instead, 2 properties critical to program correctness are:

1. Preserving exception behavior - any changes in instruction execution order must not change how exceptions are raised in program (no new exceptions)
   Example:
   ```
   DADDU R2,R3,R4
   BEQZ R2,L1
   LW R1,0(R2)
   L1:
   ```
   Problem with moving LW before BEQZ? Note: no data dependence between load and branch, only control dependence!

2. Data flow: actual flow of data values among instructions that produce results and those that consume them - branches make flow dynamic, determine which instruction is supplier of data
   Example:
   ```
   DADDU R1,R2,R3
   BEQZ R4,L
   NOP
   DSUBU R1,R5,R6
   L:
   OR R7,R1,R8
   ```
   OR depends on DADDU or DSUBU?

Must preserve data flow on execution Sometimes violating control dependence cannot affect either exception behaviour or data flow suppose R4 is unused after SKIPNEXT
Example:
```
DADDU R1,R2,R3
BEQZ R12,SKIPNEXT
DSUBU R4,R5,R6
DADDU R5,R4,R9
SKIPNEXT:
OR R7,R8,R9
```
Can we now move DSUBU to before branch? Yes
”liveness” - will value be used by an upcoming instruction ?
”speculation”

12.4 Advantages of Dynamic Scheduling

- Handles cases when dependences unknown at compile time (e.g., because they may involve a memory reference)

- It simplifies the compiler
- Allows code that compiled for one pipeline to run efficiently on a different pipeline
- Hardware speculation, a technique with significant performance advantages, builds on dynamic scheduling
- Complex in HW

HW Schemes: Instruction Parallelism

- Key idea: Allow instructions behind stall to proceed
  ```
  DIV.D F0,F2,F4
  ADD.D F10,F0,F8
  SUB.D F12,F8,F14
  ```
- Enables out-of-order execution and allows out-of-order completion
- Will distinguish when an instruction begins execution and when it completes execution; between 2 times, the instruction is in execution
- In a dynamically scheduled pipeline, all instructions pass through issue stage in order (in-order issue)
- Remember: out-of-order execution complicates exception behavior precise exceptions
  - pipeline may have already *completed* instructions that are *later* in the program order than the instr causing the exception
  - pipeline may have *not yet completed* instructions that are *earlier* in the program order than the instr causing the exception

Danger: WAR and WAW hazards
```
DIV.D F0,F2,F4
ADD.D F6,F0,F8
S.D F6,0(R1)
SUB.D F8,F10,F14
MUL.D F6,F10,F8
```
antidependence ADD and SUB
output dependence ADD and MUL
Suppose S and T are temporary registers no more dependences:
```
DIV.D F0,F2,F4
ADD.D S,F0,F8
S.D S,0(R1)
SUB.D T,F10,F14
MUL.D F6,F10,T
```

12.4.1 Dynamic Scheduling Step 1

- Simple pipeline had 1 stage to check both structural and data hazards: Instruction Decode (ID), also called Instruction Issue
- Split the ID pipe stage of simple 5-stage pipeline into 2 stages:

- *Issue* Decode instructions, check for structural hazards
- *Read operands* Wait until no data hazards, then read operands
- Istruction fetch may fetch either into instruction register or instruction queue

12.5 A Dynamic Algorithm: Tomasulo's Algorithm

- Goal: High Performance without special compilers
- Small number of floating point registers (4 in 360) prevented interesting compiler scheduling of operations
  - This led Tomasulo to try to figure out how to get more effective registers
  - renaming in hardware!
- RAW hazards:
  - execute instr only if both operands available
- WAR, WAW:
  - arise from name dependences
  - register renaming

Hazard detection, execution control and buffers distributed with Function Units (FU)
- FU buffers called "reservation stations"; have pending operands Registers in instructions replaced by values or pointers to reservation stations(RS);
- form of register renaming ;
- avoids WAR, WAW hazards ;
- More reservation stations than registers, so can do optimizations compilers can not.
Results to FU from RS, not through registers, over Common Data Bus that broadcasts results to all FUs Load and Stores treated as FUs with RSs as well.
If successive writes to same register overlap in execution, only last one is actually used to update register.
Integer instructions can get ahead and go past branches, allowing FP ops beyond basic block in FP queue.

12.5.1 RS contains:

- operation to perform
- operands if they are available
- or references to RSs that produce the operand, if not - load and store buffers - hold data or addresses coming from or going to memory - treated like FUs

12.5.2 Reservation Station Components

Op: Operation to perform in the unit (e.g., + or)
Vj, Vk: Value of Source operands

- Store buffers has V field, result to be stored Qj, Qk: Reservation stations producing source registers (value to be written)
- Note: Qj,Qk=0 are ready
- Store buffers only have Qi for RS producing result Busy: Indicates reservation station or FU is busy A: Address

Register result status indicates which functional unit will write each register, if one exists. Blank when no pending instructions that will write that register.

12.5.3 Three Stages of Tomasulo Algorithm

(may take arbitrary number of cycles)

1. Issue get instruction from FP instr Queue If reservation station free (no structural hazard), control issues instr and sends operands (renames registers).
2. Execute operate on operands (EX) When both operands ready then execute; if not ready, watch Common Data Bus for result
3. Write result finish execution (WB) Write on Common Data Bus to all awaiting units; mark reservation station available
    - Normal data bus: data + destination (”go to” bus)
    - Common data bus: data + source (”come from” bus)
      - 64 bits of data + 4 bits of Functional Unit source address
      - Write if matches expected Functional Unit (produces result)
      - Does the broadcast
    - Example speed: 3 clocks for floating point, +,-;
      10 for * ;
      40 clks for /

12.5.4 Tomasulo Disadvantages

- Complexity
  - design delays of 360/91, MIPS 10000, Alpha 21264, IBM PPC 620, but not in silicon!
- Many associative stores (CDB) at high speed
- Performance limited by Common Data Bus
  - Each CDB must go to multiple functional units
  - high capacitance, high wiring density
  - Number of functional units that can complete per cycle limited to one! Multiple CDBs more FU logic for parallel assoc stores
- Non-precise interrupts!

Memory Hazards

- With the Tomasulo algorithm, memory loads and stores can complete in any order, provided that:
  - Only one processor in system using those memory addresses
  - All loads and stores are to different addresses

- To avoid RAW, WAR, WAW hazards with a single processor,memory operations to the same address must occur in program order

- Simple resolution, check addresses of previously issued loads and stores
  - Loads from same memory address must wait for existing stores (RAW - use bypassing , more in chapter 5)
  - Stores must wait for existing stores and loads to same address

Why can Tomasulo overlap iterations of loops?

- Register renaming
  - Multiple iterations use different physical destinations for registers (dynamic loop unrolling).

- Reservation stations
  - Permit instruction issue to advance past integer control flow operations
  - Also buffer old values of registers - totally avoiding the WAR stall that we saw in the scoreboard.

- Other perspective: Tomasulo building data flow dependency graph on the fly.

Two major advantages of Tomasuloś scheme:

1. the distribution of the hazard detection logic - distributed reservation stations and the CDB
   - If multiple instructions waiting on single result, each instruction has other operand, then instructions can be released simultaneously by broadcast on CDB
   - If a centralized register file were used, the units would have to read their results from the registers when register buses are available.
2. the elimination of stalls for WAW and WAR hazards

What about Precise Interrupts?

- State of machine looks as if no instruction beyond faulting instructions has issued

- Tomasulo had: In-order issue, out-of-order execution, and out-of-order completion

- Need to ”fix” the out-of-order completion aspect so that we can find precise breakpoint in instruction stream, and provide single PC to return to.

12.5.5 Tomasulo Summary

- Reservations stations: implicit register renaming to larger set of registers + buffering source operands
  - Prevents registers as bottleneck
  - Avoids WAR, WAW hazards of Scoreboard
  - Allows loop unrolling in HW

- Not limited to basic blocks (integer units gets ahead, beyond branches)
- Today, helps cache misses as well
  - Do not stall for L1 Data cache miss (insufficient ILP for L2 miss?)
- Lasting Contributions
  - Dynamic scheduling
  - Register renaming
  - Load/store disambiguation

13 Branch prediction, speculation, multiple issue, limitations to ILP

Name: Maarten van Casteren
Studentno.: 0232823
LIACS email: mvcaster@liacs.nl

13.1 Speculation

13.1.1 Relationship between precise interrupts and speculation

- Speculation means: guess an outcome and check whether that outcome was correct.
- This is important for branch prediction: We need to ”take our best shot” at predicting the branch direction.
- If we speculate and make a wrong guess, we need to back up and restart execution at the point where we predicted the branch incorrectly. This is exactly the same as precise exceptions, covered in a previous section.
- The technique we use for both precise interrupts/exceptions and speculation is: In-order completion or commit, but out-of-order execution.

13.1.2 Hardware support for precise interrupts

We need a Hardware buffer for the results of uncommitted instructions: A Reorder Buffer.

- The Reorder buffer has 4 main fields: instruction, destination, value, ready
- When execution of an instruction completes, we use a reorder buffer number instead of a reservation station.
- The reorder buffer can supply operands between the completing of an instruction, and the committing.
- This means that you have more registers (for automatic renaming) like the reservation stations.
- Instructions commit. Once an instruction commits, the result is put into the register.
- The result is that it is easy to undo speculated instructions on mispredicted branches, or exceptions: Simply flush the reorder buffer.

13.1.3 Four Steps of Speculative Tomasulo Algorithm

1. Issue - Get the instruction from the Floating Point Op Queue.
   If a reservation station and a reorder buffer slot are free, issue the instruction and send the operands and the reorder buffer number for the destination.
2. Execution - Operate on operands (EX)
   When both operands are ready, then execute the instruction. If not ready, watch the Common Data Bus for the result. When both operands are in the reservation station, execute. This eliminates RAW hazards.

3. Write result - Finish execution (WB)
   Write on Common Data Bus to all awaiting Floating Point Units and also to the reorder buffer. Mark reservation station as available.

4. Commit - Update register (or memory) with reorder result
   When the instruction at the head of the reorder buffer and it's result are present, update the corresponding register with the result (or store to memory) and remove the instruction from the reorder buffer. A mispredicted branch flushes the reorder buffer.

Speculation with a Reorder Buffer allows execution past a branch, and then discard if the prediction fails. Instructions are held in the buffer until the branch can commit.

13.1.4 Tomasulo Summary

- Reservations stations: implicit register renaming to larger set of registers + buffering source operands
  - Prevents registers as bottleneck
  - Avoids WAR, WAW hazards of Scoreboard
  - Allows loop unrolling in HW
- Not limited to basic blocks (integer units gets ahead, beyond branches)
- Today, helps cache misses as well
  - Do not stall for L1 Data cache miss (insufficient ILP for L2 miss?)
- Lasting Contributions
  - Dynamic scheduling
  - Register renaming
  - Load/store disambiguation

13.1.5 Case for Branch Prediction when Issue N instructions per clock cycle

- Branches will arrive up to N times faster in an N-issue processor
- Amdahl's Law implicates that the relative impact of the control stalls will be larger with the lower potential CPI in an N-issue processor

13.2 Dynamic Branch Prediction

13.2.1 Goal of Branch Prediction

- Allow the processor to resolve the outcome of a branch early.
- It is good for CPI = 1, but crucial for CPI ¡ 1.
- So far, we have studied static schemes: Predict not-taken and delayed branches.
- The effectiveness of branch prediction depends on:
  - accuracy
  - cost of branch when well-predicted
  - cost of branch when mis-predicted

- Of course, prediction is only useful if determining the branch outcome takes longer than calculating the Program Counter.

13.2.2 7 Branch Prediction Schemes:

1. 1-bit Branch-Prediction Buffer (or Branch History Table)
2. 2-bit Branch-Prediction Buffer (or Branch History Table)
3. Correlating Branch Prediction Buffer
4. Tournament Branch Predictor
5. Branch Target Buffer
6. Integrated Instruction Fetch Units
7. Return Address Predictors

13.2.3 Branch Prediction Buffer (or Branch History Table, or BHT)

- The lower bits of the Program Counter addresses become an index table of 1-bit values. This bit says whether or not the branch was taken last time. No complete address check is done. This saves hardware, but may result in misprediction because it was the wrong branch.
- Problem: in a loop, a 1-bit BHT will cause 2 mispredictions (avg is 9 iterations before exit):
  - End of loop case, when it exits instead of looping as before
  - First time through loop on next time through code, when it predicts exit instead of looping
  This results in only 80% accuracy even if you loop 90% of the time.

Solution: use a 2-bit scheme where the prediction only changes if you mispredict twice.

13.2.4 Correlating Branches

The idea of Correlating Branches is that the outcome of recently executed branches is related to the behavior of the next branch (as well as the history of that branch behavior). We keep a 2 bit global BHT (which tracks the outcome of the last 2 branches before this branch), plus a 2 bit local BHT which tracks the outcome of the current branch for each combination of the global BHT.

13.2.5 Tournament Predictors

- The motivation for correlating branch predictors is that a 2-bit predictor failed on important branches. By adding global information, performance is improved.
- Tournament predictors use 2 predictors, 1 based on global information and 1 based on local information, and are combined with a selector.
- The hope is that we select the right predictor for the right branch.

13.2.6 Tournament Predictor in Alpha 21264

Working of the tournament predictor in the Alpha 21264:

- 4K 2-bit counters to choose from among a global predictor and a local predictor (indexed by address)
- The global predictor also has 4K entries and is indexed by the history of the last 12 branches; each entry in the global predictor is a standard 2-bit predictor.
  - 12-bit pattern: if the i-th bit is 0, this means that the i-th prior branch was not taken. If the i-th bit is 1, this means that the i-th prior branch was taken;
- The local predictor consists of a 2-level predictor:
  - The top level is a local history table consisting of 1024 10-bit entries; each 10-bit entry corresponds to the most recent 10 branch outcomes for the entry. 10-bit history allows patterns of 10 branches to be discovered and predicted.
  - In the second level, the selected entry from the local history table is used to index a table of 1K entries consisting of 3-bit saturating counters, which provide the local prediction
- The total size: 4K*2 + 4K*2 + 1K*10 + 1K*3 = 29K bits!

13.2.7 Need Address at Same Time as Prediction

For this problem we need a Branch Target Buffer. This buffer stores the address of the branch index to get the prediction AND branch address. (if the branch is taken)

13.2.8 Getting CPI ¡ 1: Issuing Multiple Instructions/Cycle

There are a number of solutions for issuing multiple instructions per clock cycle:

- Vector Processing: Explicit coding of independent loops as operations on large vectors of numbers.
- Superscalar: A varying number of instructions/cycle (1 to 8), scheduled by compiler or by Hardware. (Tomasulo)
- (Very) Long Instruction Words (V)LIW: A fixed number of instructions (4-16) scheduled by the compiler; put operations into wide templates.

The superscalar MIPS can issue 2 instructions per clock cycle: 1 Floating Point instruction, and 1 instruction of any type.

- We fetch 64 bits/clock cycle, Int instruction on the left, FP instruction on the right.
- We can only issue the 2nd instruction if the 1st instruction issues.
- More ports are needed for FP registers to do a FP load and a FP op in a pair.

A 1 cycle load delay expands to 3 instructions in a Superscalar processor. The FP instruction in the right half can?t use it, nor instructions in next slot.

13.2.9 Multiple Issue Issues

An issue packet is a group of instructions from the fetch unit that could potentially issue in 1 clock cycle.

- If an instruction causes a structural hazard or a data hazard either due to an earlier instruction in execution or to an earlier instruction in issue packet, then that instruction does not issue.
- This results in 0 to N instruction issues per clock cycle, for an N-issue processor.

Performing issue checks in 1 cycle could limit clock cycle time: O($n^2 - n$) comparisons

- Because of that, the issue stage is usually split and pipelined.
- The 1st stage decides how many instructions from within this packet can issue, 2nd stage examines hazards among the selected instructions and those that have already been issued.
- This leads to higher branch penalties, which means that prediction accuracy is important.

13.2.10 Multiple Issue Challenges

While splitting Integer and Floating Point instructions is simple for the Hardware, we can get a CPI of 0.5 only for programs with exactly 50% FP instructions and no hazards.
If more instructions issue at same time, the decode and issue stages become more and more difficult.

13.2.11 Dynamic Scheduling in Superscalar: The easy way

How to issue two instructions and keep in-order instruction issue for Tomasulo?

- Assume 1 integer and 1 floating point operation
- 1 Tomasulo control for integer operations, 1 for floating point.

Issue at 2X Clock rate, so that issue remains in order.
Only loads and stores might cause a dependency between an integer and FP issue:

- Replace the load reservation station with a load queue; operands must be read in the order they are fetched.
- A load checks addresses in the Store Queue to avoid a Read After Write violation.
- A store checks addresses in the Load Queue to avoid a Write After Read or Write After Write violation.

13.2.12 Register renaming, virtual registers versus Reorder Buffers

An alternative to a Reorder Buffer is a larger virtual set of registers and register renaming.
Virtual registers hold both architecturally visible registers and temporary values.

- This replaces the functions of the reorder buffer and reservation station.

The renaming process maps names of architectural registers to registers in the virtual register set.

- A changing subset of virtual registers contains the architecturally visible registers.

This simplifies the committing of an instruction: Mark the register as a no longer speculative, free register with an old value.
Register renaming can add 40-80 extra registers.

- The size limits the number of instructions in execution. (registers are being used until a commit)

13.2.13 How much to speculate?

- Speculation Pro: We can uncover events that would otherwise stall the pipeline. (cache misses)
- Speculation Con: Speculating is costly if an exceptional event occurs when speculation was incorrect.
- Typical solution: Speculation allows only low-cost exceptional events. (1st-level cache miss) When an expensive exceptional event occurs, (2nd-level cache miss or TLB miss) the processor waits until the instruction causing the event is no longer speculative before handling the event.
- This is easier when allowing only a single branch per cycle.
  - Newer processors speculate across multiple branches and allow multiple branches in an instruction issue.

13.2.14 Beyond ILP: Thread Level Parallellism

- There are diminishing returns for finding ILP in code that was designed to be sequential.
- To use functional units for more than one process/program, what do we need?
  - Multiple program counters.
  - Multiple register files or register renaming that is aware of multiple threads and keeps hazards independent.
- A thread may be a process in the same program sharing the same memory, or an entirely different program (e.g. multitasking server)

14 Exploiting Instruction-Level Parallelism with Software Approaches

Name: Thijs van Ommen
Studentno.: S0222704
LIACS email: mvommen@liacs.nl

14.1 Loop Unrolling

To make efficient use of a pipeline, stalls must be avoided. The compiler can help to do this by arranging instructions in such a way, that two instructions with a data dependence are seperated by enough other instructions to avoid data hazards. The extent to which this can be achieved varies, as it is limited by the amount of ILP available in the program.
Loops form one case where there is ILP for the compiler to exploit. This is done by **loop unrolling**: the loop body is copied a few times, with minor adjustments, and these copies are placed together in a new loop. This newer version spends less time on loop overhead (counter increment, conditional jump). A more important advantage is that there are now more instructions that can be freely rescheduled (assuming that consecutive iterations of the loop body were independent), sometimes eliminating all stalls.
How much can be gained from this technique is limited by:

- The amount of gain from reducing the loop overhead each iteration;
- The amount of memory available to unroll loops;
- The number of registers that can be used in the new loop.

14.2 Static Branch Prediction

If anything is known at compile time about the expectation of a given branch instruction, this knowledge can sometimes be used by the compiler to eliminate control hazards as well as data hazards between instructions near a branch and its destination.
But where does this knowledge come from? A simple answer is to simply assume that all branches are taken. A better way is to predict backward branches as taken (because they are usually loops), and forward branches as not taken. Even more accuracy can be gained by using actual data from previous runs of the program.

14.3 Static Multiple Issue: The VLIW Approach

Superscalar processors can issue multiple instructions simultaneously, either at run- or at compile time. The first approach requires expensive hardware, while the second needs a lot of compiler support.
VLIW (which stands for Very Long Instruction Word) takes the middle road: the compiler does supply some, but not all information about possible dependencies. A VLIW has several independent functional units. The instructions in a single instruction word must be independent; this responsibility is left to the compiler.

We distinguish between local and global scheduling. The first refers to techniques that affect a single basic block (a sequence of instructions without branches). Global scheduling also involves looking past branches, usually forming large basic blocks that local scheduling can then improve on.
Early VLIWs had several problems:

- Long code size, due to extensive loop unrolling and because the instruction slots in an instruction word are rarely all filled;
- Because they didn't use any hardware-based hazard detection, all instructions were kept synchronyzed. A single functional unit could stall all the others;
- Because the compilation process used a lot of information about the current VLIW, a compiled program would not operate on later versions without recompilation.

14.4 Advanced Compiler Support for Exposing and Exploiting ILP

A loop carried dependence is a data dependence between different iterations of a loop. A loop without such dependencies is said to be loop-level parallel. The iterations of a loop be scheduled freely by the compiler.
Non-parallel loop can also be rescheduled in some cases. The operations made on a simple counting variable in a for-loop can be modified by the compiler to achieve the intended result. Also, if there are dependencies in the loop, but they don't form cycles, then the loop can be made parallel.

14.4.1 Finding Dependencies

A problem with the analysis of potential dependencies is that tests only give inexact information: they either conclude that there is no dependence, or that there may be a dependence, but not generally that there is actually a dependence. High-level programming languages that allow aliases (differently-named references to a single memory location) further complicate matters, which must be resolved using points-to analysis.
There are also test that are exact, but they are restricted in the situations where that test may be performed. Sometimes tests are organized into hierarchies, with the more expensive tests applied only if simpler tests fail.
A compiler can perform points-to analysis by checking certain conditions under which two pointer can or can't point to the same object:

- Two pointers that point to different types can't point to the same object;
- A pointer can only point to an objects if that object's address can be assigned to that pointer directly (when the object is allocated or its address taken) or indirectly (pointer assignments).

As with dependence analysis, points-to analysis would be too time-consuming to perform accurately for real programs, so compilers must approximate.

14.4.2 Eliminating dependent computations

Computations can be optimized in several ways. This paragraph explains two of the more common.
Copy propagation is a technique in which operations that copy values are eliminated.
Algebraic expressions can be expressed as trees. Tree height reduction modifies these trees to allow more parallelism.

14.4.3 Symbolic Loop Unrolling

Two techniques that serve the same purpose as loop unrolling (namely, creating large basic blocks) are software pipelining and trace scheduling.
Software pipelining builds a new loop by placing instructions from several different iterations from the original loop in a single iteration. Its advantage over loop unrolling is that it takes less code, though it is normally slightly slower.

14.4.4 Global Code Scheduling

When a loop body contains internal control flow, global code scheduling is required to perform optimizations. All possible control flows must be determined, and their relative frequency is required to reschedule them efficiently. Many other factors must be taken into account as well. Several techniques have been developed to simplify the process.

14.4.5 Trace Scheduling: Focusing on the Critical Path

This technique uses two steps. The first, trace scheduling, chooses a trace: a sequence of basic blocks. Then the next step, trace compaction, consists of code scheduling with the trace to generate wide instructions.

14.4.6 Superblocks

Superblocks are like basic blocks, but allow multiple exit points. Viewing code in this way makes optimizations easier to detect. Because only one entrance point is allowed, tail duplication may be necessary.

14.5 Hardware Support for Exposing More Parallelism at Compile Time

Because software can’t always make efficient use of available ILP, hardware support is useful. Some techniques are discussed here.

14.5.1 Conditional or Predicated Instructions

By introducing instructions that will only perform their normal operation if a certain condition is true, control dependencies may be converted to data depencencies. This is especially useful in multi-issue architectures, because of the difficulties involved with issuing multiple branch instructions simulataneously.
Some limiting factors are the following:

- Instructions whose condition is false still take time, so large amounts of such instructions are less efficient;
- Evaluating the condition may cause stalls;
- Only simple conditions are available;
- Conditional instructions will be slower than conditional ones.

14.5.2 Hardware Support for Preserving Exception Behaviour

Preserving exception behaviour will limit the options to use ILP, unless there is hardware support in one of the following ways:

- Hardware and operating system ignore exeptions for speculative instructions: this only preserves exception behaviour on programs that produce no non-resumable exceptions;
- As above, but extra checks cause exception behaviour to be preserved;
- Poison bits attached to each register are set when that register is written by a speculative instruction that would have caused an exception, but the exception only occurs when a normal instruction tries to read from that register;
- Results of speculative instructions are buffered by hardware.

14.5.3 Hardware Support for Memory Reference Speculation

The compiler can't always be certain that moving around loads and stores will preserve the program's functionality, because they might or might not refer to the same memory location. Hardware can help out when a special instruction in place of the original load is introduced. If subsequent stores affect the memory address marked by the new instruction before the actual load, speculation has failed, and some measures have to be taken to correct the situation.

14.6 Crosscutting Issues: Hardware versus Software Speculation Mechanisms

This paragraph compares the two different approaches to exploiting ILP: hardware-based and software-based.

- Disambiguating memory references can't be done accurately and efficiently at compile-time;
- Dynamic branch predictors are generally better than static ones;
- Not all software-based approaches to exception handling maintain precise exception behaviour;
- Hardware-based speculation doesn't generate longer programs;
- Software-based speculation can see further and thus find better code schedules;
- Hardware-based speculation doesn't require code to be recompiled to take advantage of a newer architecture.

Both approaches have their own costs and advantages, and these have to be weighed to determine the most suitable solution. A combination is often best.

Name: Alexander.Nezhinsky
Studentno.: 0106658
LIACS email: anezhins@liacs.nl

15 Samenvatting Hoofdstuk 4

Uit hoofdstuk 3 was duidelijk dat met behulp van pipelining de instructies elkaar kunnen overlappen. Deze mogelijke overlap tussen instructies wordt ilp genoemd (instruction level parallelalism). In dit hoofdstuk bekijken we de technieken om de ideen van pipelining te onderbouwen, door het aantal parallellisme zo veel mogelijk te vergroten.
Het aantal parallellisme aanwezig in een Basic bloc is vrij weinig (als er geen branches in zitten). Om het aantal parallellisme te vergroten kunnen we bijvoorbeeld gebruik maken van loop-level parallelalism.
Om een pipeline vol te houden moeten we steeds naar instructies zoeken, die onafhankelijk van elkaar zijn en dus tegelijkertijd kunnen worden uitgevoerd. De mogelijkheid van een compiler om de van elkaar afhankelijke instructies te schedulen is afhankelijk van het aantal beschikbare ILP en van de latencys van de functional units in een pipeline. Om scheduling te verbeteren kunnen we gebruik maken van loop unrolling. Dit wordt gedaan door het body van de loop meerdere keer te kopiren. Daarna kijken we welke instructions in welke volgorde kunnen worden uitgevoerd om executie tijd te verkleinen. Voor loop unrolling gelden de volgende regels:

- kijk of het mogelijk is om SD te verplaatsen achter de SUBI en BNEZ en verander de SD offset.
- kijk of loop unrolling nuttig was en het iteractions inderdaad onafhankelijk zijn
- gebruik verschillende registers voor verschillende berekeningen
- verwijder extra tests en branches en verander loop maintenance code
- kijk of loads en stores wel omgewisseld kunnen worden
- schedule de code, behoud alleen de dependencies die voor hetzelfde resultaat zorgen als de originele code.

De dependencies kunnen vinden is belangrijk, omdat wanneer twee processen een dependency hebben (afhankelijk van elkaar zijn) ze niet parallel kunnen verlopen. En we moeten juist uitzoeken welke instructies wel parallel kunnen verlopen om ILP te benutten. Instructies die van plaats gewisseld kunnen worden zijn onafhankelijk en dus parallel.
Er zijn drie types van dependencies: data dependencies, name dependencies en control dependencies. Een instructie j is data-dependent met betrekking tot instructie i als:

- instructie i een resultaat produceert, dat gebruikt wordt door instructie j
- instructie j is data dependent met betrekking tot instructie k, en instructie k is data dependent met betrekking tot instructie i.

Er is sprake van een name dependence als twee instructies dezelfde register of geheugenlocatie gebruiken (name), maar er is geen data-flow tussen de instructies geassocieerd met deze name. We onderscheiden twee types name dependencies tussen een instructie i die aan instructie j voorafgaat:

1. antidependence (als instructie j iets schrijft naar een geheugenadres (of register), i leest en i gaat voor j (WAR hazard))
2. output dependence (als i en j naar dezelfde register (of geheugen locatie) schrijven (WAW hazard))

Er is sprake van een control dependency als een instructie afhankelijk is van een branch instructie. verder geldt voor control dependencies:

- een instructie, die control dependent is kan nooit worden verplaatst voor de branch waar deze afhankelijk van is.
- een instructie, die niet afhankelijk (dependend) is van een branch kan niet worden verplaatst naar een plaats na de branch.

Soms is het echter mogelijk om deze regels niet op te volgen en toch een correcte execution te krijgen.
Erg belangrijk voor een correct programma zijn exception behavior en data flow. Het behouden van exception behavior houdt in dat de verandering van de volgorde van de instructies niet mag leiden tot verandering van het ontstaan van exceptions. Data flow moet altijd zelfde blijven. bij branches kunnen we wel gebruik maken van speculation - we kunnen gokken of een branch wel of niet genomen wordt. hiervan kunnen we alleen gebruikmaken, als de data-flow alsnog zelfde blijft. Wanneer hardware de executie van de instructies zo hergroepeert, dat het aantal stalls vermindert, spreken we van de dynamic scheduling. dynamic scheduling heeft enkele voordelen:

- het wordt mogelijk sommige gevallen te bewerken, wanneer de dependencies nog onbekend zijn tijdens de compilatie
- het maakt de compiler simpeler
- het is mogelijk om een voor een bepaalde pipeline gecompileerd programma efficint uit te voeren met een andere pipeline

De nadelen van dynamic scheduling zijn de hoge kosten en complexe hardware.
Met dynamic schedulen maken we gebruik van out-of-order completion. Hierbij wordt de ID-pipe stage gesplitst in twee stages:

- Issue - decodeer instructies, check voor structural hazards
- read operands - wacht totdat er geen data-hazards zijn, lees daarna de operands

In een dynamically- scheduled pipeline komen alle instructies in-order in de issue stage en kunnen daarna in de read operands stage stallen of elkaar passeren (out of order). Scorebording zorgt er voor dat instructies out of order kunnen worden uitgevoerd zodra ze genoeg resources hebben en er geen data dependencies zijn. het doel van scoreboarding is dus om de instructies zo snel mogelijk uit te voeren (een instructie per clock cycle). Om out-of-order te benutten moeten meerdere instructies tegelijk in de EX-stage zijn. dit dan door meerdere functionele units te hebben, deze te pipelinen, of beide. de vier stappen die de ID, EX en WB stappen in de standaard DLX-pipeline vervangen zijn de volgende:

1. issue (controle of een functional unit vrij is en er geen hazard aanwezig is)
2. read operands (kijken of operands beschikbaar zijn)
3. Execution (als een functional unit klaar wordt dit aan scoreboard doorgegeven)
4. write result(check voor WAR hazard en stall indien nodig)

Een completing instruction mag zijn resultaat niet wegschrijven als:

- er is een instructie die hieraan voorafgaat en nog niet zijn operands heeft gelezen
- een van de operands is dezelfde register als het resultaat van deze instructie

een scoreboard bestaat uit drie onderdelen:

- Instruction status - laat zien in welke stap de instructie zich bevindt.
- functional unit status - laat de staat van de functional unit zien.
- register result status-toont welke functional unit naar elke register zal schijven.

Een scoreboard gebruikt aanwezige ILP om het aantal stalls door data dependencies te minimaliseren. het wordt daarbij gelimiteerd door de volgende factoren:

- hoeveelheid parallellisme
- aantal scoreboard entries (hoe ver vooruit kan worden gekeken)
- aantal en type van functional units
- aantal antidependencies en output dependencies (deze leiden tot WAR en WAW hazards)

een andere methode voor dynamic scheduling is het Tomasulo approach. Tomasulo's werking lijkt op scoreboard, maar heeft reservation stations om operands vast te houden. Deze reservation stations hebben tag fields, die beschrijven welke reservation station instructies hebben, die resultaat produceren voor de source operands. Ook doet Tomasulo aan register renaming (operands hernoemen naar namen van reservation stations) en zij load en store buffers aanwezig. Zo worden WAR en WAW hazards voorkomen (waardes hoeven niet naar registers te gaan, maar direct naar reservation stations). Hazard detectie is dus gedistributeerd over de reservation stations en niet gecentraliseerd, als bij scoreboard. We bekijken de stappen waar een instructie langs komt:

1. Issue (als operation een FP is, wordt deze naar een reservation station gestuurd, registers worden hernoemd)
2. Executeer (als alle operands beschikbaar zijn, in een reservation station plaatsen)
3. Write result

Een nadeel van Tomasulo is het complexe schema ervan en het aantal benodigde hardware.
Dynamic Hardware prediction houdt zich bezig met het voorspellen of een branch wel of niet wordt genomen. De simpelste branch-predicting schema is een branch-prediction buffer oftewel een branch history table. Deze onthoudt of een branch de vorige keer wel of niet genomen was. 2-bit prediction buffers kijken of de branch 2 keer achter elkaar is genomen (of niet) en dan pas wordt er verwacht dat dat ook de derde keer gebeurt. als de voorspelling verkeerds was wordt de table gepdate.
Soms is het gedrag van bepaalde branches afhankelijk van het gedrag van andere branches.
Er is sprake van eer prediction en een correlation bit, als een bit wordt gemerkt wanneer de branch wel is genomen en de andere als de branch niet is genomen.
Het aantal bits in een (m,n) predictor is 2exp m * n * aantal prediction entries geselecteerd door branch adres.
Bij Branch target buffer slaan we niet de richting van de branch, maar de PC waar naar toe gebranched wordt. Branch target buffer is een groot associatief geheugen voor (branch van PC, branch naar PC) paren.Tijdens de IF fase wordt er in de tabel van branch target buffer gekeken of de PC er instaat, is dat het geval, dan wordt er gegaan naar de ”branch naar PC”.
In de vorige twee paragrafen werden technieken getoond om data en control stalls te elimineren en zo een ideale CPI van 1 te verkrijgen. Als we echter de CPI lager dan een willen krijgen moeten meerdere instructies tegelijk kunnen worden uitgevoerd in een clock cycle. Er is dan sprake van multiple-issue processors. er zijn twee types multiple-issue processors:

- superscalar processors (varierend aantal instructies per clock cycle) (static scheduled en dynamically scheduled)
- VLIW (very long instruction word) (vast aantal instructies per clock cycle) processors

We bekijken eerst statically scheduled superscalars. Hier wordt de compiler gebruikt om instructies te schedulen. De compiler moet naar onafhankelijke instructies zoeken en deze samen in het instruction stream zetten (zodat ze parallel kunnen worden uitgevoerd). De opkomende problemen met deze methode zijn:

- FP units moeten pipelined worden of er moeten meerdere onafhankelijke units van worden gemaakt (anders is FP unit een bottleneck)
- als er een FP-load, FP-store of FP-move instructie aanwezig is, zijn er extra poorten nodig in het FP register (anders wordt dit een bottleneck)
- loads hebben een latency van 1 en kunnen dus niet gelijk worden gebruikt (voor branch geldt dit ook)
- meer complexe instruction decoding is benodigd.

Dynamically scheduled superscalar is te zien als een extensie op het Tomasulo algoritme. Een floating-point instructie en een integer worden tegelijk naar hun reservation stations gestuurd. instructies worden altijd in-order uitgevoerd. Om een probleem op te lossen wanneer er twee van elkaar afhankelijke instructies tegelijk naar de reservation stations worden gestuurd (kunnen dus niet tegelijkertijd worden uitgevoerd) zij er twee oplossingen:

- er wordt aangenomen dat de issue stage van de pipeline 2 keer zo snel kan worden uitgevoerd en zo 2 instructies na elkaar worden uitgevoerd

- gebruik een rij om het resultaat van een FP load en move op te slaan.

We bespreken compiler technologie om de hoeveelheid parallellisme in een algoritme te vergroten. Het vinden van afhankelijkheden in een programma gaat als volgt:

- goede scheduling van de orde
- bepalen welke loops parallellisme kunnen bevatten
- elimineren naam dependencies

Het zoeken naar dependencies wordt des te moeilijker bij talen als C, omdat deze arrays en pointers bevatten.
Er is sprake van een recurrence, wanneer een variabele gedefinieerd is, gebaseerd op de waarde van deze variabele in een eerdere iteratie.Vaak is het de iteratie, die er gelijk voor staat.
Het is om 2 redenen belangrijk om recurrentie te vinden:

1. sommige systemen hebben speciale ondersteuning om recurrenties uit te voeren.
2. sommige recurrenties kunnen een zijn voor een grote hoeveelheid parallelalisme.

Hoe groter het dependence distance, hoe meer parallelalisme we kunnen verkrijgen door loop unrolling.
Een array index wordt affine genoemd als het geschreven kan worden in de vorm a * i + b (a en b constanten en i index variabele(loop)). Om dependency vinden moet er gekeken worden of twee functies affine zijn en dezelfde waarde bevatten voor verschillende indices tussen de loop bounds.
Het is meestal niet mogelijk om zo een dependence te vinden tijdens compilatie. Dit komt doordat de benodigde waarden vaak onbekend zijn.
De situaties, wanneer dependence analysis niet de juiste informatie kan verschaffen:

- als objecten met pointers zijn gerefereerd en niet met array indices.
- wanneer array indexing via een andere array is gedaan
- als dependence bestaat voor bepaalde waarden van de input, maar eigenlijk niet bestaan, wanneer de code wordt uitgevoerd

Software pipelining is een techniek om loops te herorganiseren. Deze techniek kiest instructies van verschillende loop-iteraties en splitst zo de afhankelijke instructies binnen een iteratie van de originele loop.
Trace scheduling breidt loop unrolling uit door het gebruik van een techniek, dat parallelalisme zoekt over conditional branches. Het is nuttig voor processors met een hoog aantal missies per clock, waar unrolling niet sufficint genoeg is om zoveel ICP te verkrijgen, dat de processor bezig blijft. Trace scheduling is een combinatie van twee processen:

- trace selection probeert een waarschijnlijke volgorde van blokken te vinden, de operaties waarvan in een kleiner aantal instructies worden gezet. Deze opeenvolging wordt een trace genoemd. Omdat loop branches met een grote waarschijnlijkheid worden genomen, genereert loop unrolling lange traces.

- trace compaction volgt trace selection op en probeert een trace in een kleiner aantal wijde instructies te krijgen. Er wordt geprobeerd om operands zo vroeg mogelijk in een sequence tree te verplaatsen en in zo weinig mogelijk wijde instructies in te pakken. Data dependencies worden voorkomen door loop unrolling en dependence analysis te gebruiken om te kijken of twee referenties naar dezelfde adres verwijzen.

Trace scheduling zorgt ervoor, dat als de voorspelling correct is de compiler kan bepalen of deze code waarschijnlijk zal leiden tot een snellere code.
Wanner we het gedrag van branches niet weten zijn alleen de compilertechnieken niet genoeg om veel ILP te verkrijgen. Om branches te elimineren en de compiler te helpen instructies voorbij de branches te verplaatsen, wordt gebruik gemaakt van conditional of predicted instructions. Deze hebben echter ook limitaties. Om nog meer parallelalisme te verkrijgen is speculation mogelijk. Speculation staat toe de instructie uit te voeren, voordat we weten, of deze wel moet worden uitgevoerd.
Het concept van conditional of predicted instructions is het volgende:

- als condition true is wordt de instructie normaal uitgevoerd
- als condition false is wordt de instructie als een no-op beschouwd.

Voordelen van conditional of predicted instructions zijn dat ze soms een branch kunnen elimineren en mogelijk het gedrag van het pipeline verbeteren. conditional instructions kunnen ook scheduling verbeteren van superscalar of VLIW processors door het gebruik van speculation. Het is belangrijk dat een conditional instruction geen exceptions genereert.
Het gebruik van conditional instructions is gelimiteerd:

- wanneer condition false is gaat alsnog tijd verloren
- vooral nuttig wanneer de condition vroeg kan worden gevalueerd
- gebruik van conditional instructions is gelimiteerd wanneer de control flow ingewikkeld is
- conditional instructions zorgen voor een iets lagere snelheid dan unconditional instructions

Er zijn de volgende methoden om meer ambitieuze speculation te ondersteunen zonder uitzonderingen (exceptions) te creren:

- hardware en operating system negeren excepties voor speculatieve instructies
- status bits worden naar het register geschreven waar het resultaat naar toe gaat van een speculated instruction. Als een normale instructie dit register wil gebruiken zorgen de poison bits voor een fout.
- mechanisme geeft door of een instructie speculative is en het hardware buffert de instructie, totdat deze niet meer speculative is.

Er zijn meer voordelen van hardware-based speculation dan software-based speculation, maar het heft een nadeel het is een complex proces en heeft veel hardware resources nodig.
Here is an example of how an encapsulated postscript file can be included: