TDTS08: whattolearn

Name	PIN	Email
Pål Kastman	851212-7575	palka285@student.liu.se
Alexander Yngve	930320-6651	aleyn573@student.liu.se

Contents

1 Introduction

This documents only purpose is to document what I will need to learn in the course tdts08.
YNGVE OM DU VILL BYTA RUBRIKERNA SÅ ÄR DET HELT OKEJ, DOM ÄR BARA DÄR SOM STÖD NU

2 Lecture 2: Memory Systems

Sequential access: Memory is organized into units of data, called records. Access must be made in a specific linear sequence. Stored addressing information is used to separate records and assist in the retrieval process. A shared read–write mechanism is used, and this must be moved from its current location to the desired location, passing an rejecting each intermediate record. Thus, the time to access an arbitrary record is highly variable. **Tape units** are sequential access.

Direct access: As with sequential access, direct access involves a shared read–write mechanism. However, individual blocks or records have a unique address based on physical location. Access is accomplished by direct access to reach a general vicinity plus sequential searching, counting, or waiting to reach the final location. Again, access time is variable. **Disk units** are direct access.

Random access: Eacha ddressable location inm emory has a unique, physically wired-in addressing mechanism. The time to access a given location is independent of the sequence of prior accesses and is constant. Thus, any location can be selected at random and directly addressed and accessed. Main memory and some cache systems are random access.

Associative: This is arandom access type of memory that enables one to make a comparison of desired bit locations within a word for a specified match, and to do this for all words simultaneously. Thus, a word is retrieved based on a portion of its contents rather than its address.

Access time (latency): For random-access memory, this is the time it takes to perform a read or write operation, that is, the time from the instant that an address is presented to the memory to the instant that data have been stored or made available for use. For non-random-access memory, access time is the time it takes to position the read–write mechanism at the desired location.

Transfer rate: This is the rate at which data can be transferred into or out of a memory unit. For random-access memory, it is equal to 1/(cycle time). For non-random-access memory, the following relationship holds:

$$T_n = T_A + \frac{n}{R}$$

T_n = Average time to read or write n bits
T_A = Average access time

n = Number of bits
R = Transfer rate, in bits per second (bps)

2.1 Memory hierarchy

Registers Built into the processors, very small and expensive.
Main Memory Memory that the cache collects data from.
Secondary Memory (of direct access type) Bigger in size, but less expensive per Byte.
Secondary Memory (of archive type) And so on.

2.2 Cache memory

There is a big gap in speed between register and main memory, this is why we need to use cache memory.

Cache memory is designed to combine the memory access time of expensive, high-speed memory combined with the large memory size of less expensive, lower-speed memory. ller, faster cache memory.

The cache contains a copy of portions of main memory. When the processor attempts to read a word of memory, a check is made to determine if the word is in the cache. If so, the word is delivered to the processor. If not, a block of main memory, consisting of some fixed number of words, is read into the cache and then the word is delivered to the processor.

We would like the size of the cache to be small enough so that the overall average cost per bit is close to that of main memory alone and large enough so that the overall average access time is close to that of the cache alone.

2.2.1 Locality of reference

The intermediate-future memory access will usually refer to the same word or words in the neighborhood, and will not have to involve the main memory.

Temporal Locality – If an item is referenced, it will tend to be referenced again in the near future.
Spatial Locality – If an item is referenced, items whose addresses are close by will tend to be referenced soon.

2.2.2 Replacement methods

yes, there's different kinds of them.

2.2.3 Cache Design

One, two, three level caches, how does it work. up- and downsides of them.

Split Caches means that we have separate caches for instructions and data.
+ Competition for the cache between instruction and data is eliminated
+ Instruction fetch can proceed in parallel with memory access from the CPU for operands.
- One may be overloaded while the other is under utilized.

Unified Caches means that both instructions and data uses the same cache.
+ Better balance the load between instruction and data fetches depending on the dynamics of the program execution.
+ Design and implementation are cheaper.
– Lower performance.

2.2.4 Direct mapping cache

Each block of the main memory is mapped into a fixed cache slot.
+ Simple to implement and therefore inexpensive.
– If a program accesses 2 blocks that map to the same cache slot repeatedly, cache miss rate is very high.

2.2.5 Associative mapping cache

A main memory block can be loaded into any slot of the cache. To determine if a block is in the cache, a mechanism is needed to simultaneously examine every slot's tag. In this case, the cache control logic interprets a memory address simply as a Tag and a Word field. The Tag field uniquely identifies a block of main memory. To determine whether a block is in the cache, the cache control logic must simultaneously examine every line's tag for a match.

The downside of this is the complex circuitry needed to examine all the tags.

2.2.6 Set Associative mapping cache

A fully associative mapped memory can be built but is very expensive, and therefore is not done. Instead we use a **set associative** cache memory.
Here the cache are divided into a number of sets, where each set contains a number of slots (just like a regular cache).

2.2.7 Fully Associative Organization

2.2.8 Write Policy

When a block in the cache is to be replaced, we have to consider two cases. If the block in the case has **not been altered**, then it can be replaced with a new block without saving the block to the main memory. Or if it **has been altered** (one block is enough), then we need to write that block back to main memory.

There are two cases to consider

1. More than one device may have access to main memory.

   For example, an I/O module may be able to read-write directly to memory. If a word has been altered only in the cache, then the corresponding memory word is invalid. Further, if the I/O device has altered main memory, then the cache word is invalid.

2. Multiple processors are attached to the same bus and each processor has its own local cache. Then, if a word is altered in one cache, it could conceivably invalidate a word in other caches.

Write through all write operations are made to main memory as well as to the cache, ensuring that main memory is always valid. Any other processor–cache module can monitor traffic to main memory to maintain consistency within its own cache. The main disadvantage of this technique is that it generates substantial memory traffic and may create a bottleneck.

Write through with buffered write The same as write-through, but instead of slowing the processor down by writing directly to main memory, the write address and data are stored in a high-speed write buffer; the write buffer transfers data to main memory while the processor continues its task.

Write back With write back, updates are made only in the cache. When an update occurs, a dirty bit, or use bit, associated with the line is set. Then, when a block is replaced, it is written back to main memory if and only if the dirty bit is set.

2.3 Virtual memory

Almost all nonembedded processors, and many embedded processors, support virtual memory. In essence, virtual memory is a facility that allows programs to address memory from a logical point of view, without regard to the amount of main memory physically available. When virtual memory is used, the address fields of machine instructions contain virtual addresses. For reads to and writes from main memory, a hardware memory management unit (MMU) translates each virtual address into a physical address in main memory.

2.3.1 Paging

Unequal fixed-size and variable-sized partitions of memory are inefficient, we need to make sure that we use the most of the memory available in the best way, we try to do this using pages.

We first divide programs into fixed sized memory blocks called pages, then we divide the main memory into equal fixed sized blocks, called page frames.

A process then needs a set of free page frames assigned by the OS. Then its up to the OS to make sure it keeps track of the free available frames, it also uses a page table to keep track of the mapping between pages and page frames.

2.3.2 Page fault

When we try to access a piece of memory that is not currently in the main memory, but instead in the virtual memory. The OS must then load the page into the main memory firs, this is called a **Page fault**.

2.3.3 Page replacement

When a page fault occurs and all the page frames are occupied, one of them must be replaced. If the page that are to be replaced, has been modified, we need to write that memory back to the secondary storage.

We want to replace a page that we think will not be accessed for the longest amount of time. The problem is that we can't predict the future, so we have to predict the future, we can do this in some different ways, described below.

2.3.4 Replacement algorithms

Not needed with direct mapping, but with associative mapping we need one of these algorithms.

First-in-first-out
Least recently used (LRU): replaces the cache that has been in the cache the longest without being referenced.
Least frequently used (LFU): replaces the cache that has been referenced the least frequent.
Random

3 Lecture 3: Instruction pipelining

The idea is to divide the workload into different stages, so that we don't have to wait for an instruction to complete before we fetch the next one.

The typical pipeline has **six** stages:
1. Fetch Instruction (FI): Fetch the instruction.
2. Decode Instruction (DI): Determine the op-code and the operand specifiers.
3. Calculate Operands (CO): Calculate the effective addresses.
4. Fetch Operands (FO): Fetch the operands.
5. Execute Instruction (EI): perform the operation.
6. Write Operand (WO): store the result in memory.

The ideal case gives us a speed-up of six times. But in practice we have some problems.

In general, a larger number of stages gives better performance.
However, a large number also increases the complexity. We have to move a lot of information between stages, and synchronize the stages. We call the problems that occur **Pipeline Hazards**.

3.1 Pipeline hazards

There are different kinds of pipeline hazards, first we have the case with dependency, A pipeline hazard occurs when the pipeline, or some portion of the pipeline, must stall because conditions do not permit continued execution. Such a pipe- line stall is also referred to as a pipeline bubble. There are three types of hazards: resource, data, and control.

3.1.1 Structural (resource) hazards

A **Resource hazards** resource hazard occurs when two (or more) instructions that are already in the pipeline need the same resource. The result is that the instructions must be executed in serial rather than parallel for a portion of the pipeline. A resource hazard is sometime referred to as a structural hazard.

3.1.2 Data hazards

A **Data Hazard** is when two instructions in a program are to be executed in sequence and both access a particular memory or register operand. If the two instructions are executed in strict sequence, no problem occurs. However, if the instructions are executed in a pipeline, then it is possible for the operand value to be updated in such a way as to produce a different result than would occur with strict sequential execution.

There are three types of data hazards:
• Read after write (RAW), or true dependency: An instruction modifies a register or memory location and a succeeding instruction reads the data in that memory or register location. A hazard occurs if the read takes place before the write operation is complete.
• Write after read(WAR), or antidependency: Aninstructionreadsaregisteror memory location and a succeeding instruction writes to the location. A hazard occurs if the write operation completes before the read operation takes place.
• Write after write (WAW), or output dependency: Two instructions both write to the same location. A hazard occurs if the write operations take place in the reverse order of the intended sequence.

We can handle this by using a technique called forwarding (bypassing). This works in the following way:
The ALU passes its result back into a MUX. If this detects that the value have been updated and not yet written back into the main memory, the value from the ALU will be used instead of the value we fetched from the memory as can be seen in figure 1.

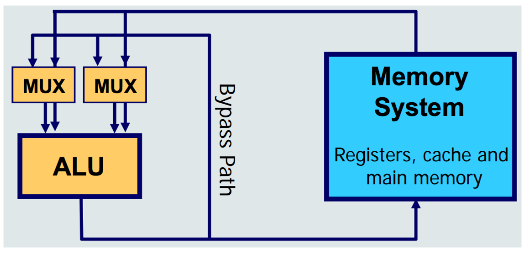


Figure 1 – The ALU passes its calculated value back into a MUX.

3.1.3 Control hazards

A **control hazard**, also known as a branch hazard, occurs when the pipeline makes the wrong decision on a branch prediction and therefore brings instructions into the pipeline that must subsequently be discarded as seen in figure 2.

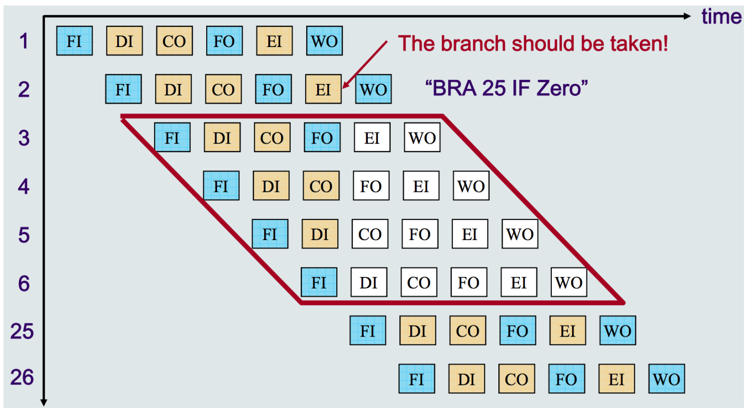


Figure 2 – Control hazards.

3.2 Branch handling

There are different ways of handling branches.

3.2.1 Stop the pipeline

We can stall all the next instructions until the current branch reaches its final stage and we know what to do, this is expensive due to that 20-35% of the things instructions executed are executed as branches.

3.2.2 Multiple streams

Implement hardware resources so that we can execute both alternatives in parallel, there are two problems with this though:

1. With multiple pipelines there are contention delays for access to the registers and to memory.

2. Additional branch instructions may enter the pipeline (either stream) before the original branch decision is resolved. Each such instruction needs an addi- tional stream.

3.2.3 Pre-fetch branch target

When a conditional branch is recognized, the target of the branch is prefetched, in addition to the instruction following the branch. This target is then saved until the branch instruction is executed. If the branch is taken, the target has already been prefetched.

3.2.4 Loop buffer

A loop buffer is a small, very-high-speed memory maintained by the instruction fetch stage of the pipeline and containing the n most recently fetched instructions, in sequence. If a branch is to be taken, the hardware first checks whether the branch target is within the buffer. If so, the next instruction is fetched from the buffer. The loop buffer has three benefits:

1. With the use of prefetching, the loop buffer will contain some instruction sequentially ahead of the current instruction fetch address. Thus, instructions fetched in sequence will be available without the usual memory access time.
2. If a branch occurs to a target just a few locations ahead of the address of the branch instruction, the target will already be in the buffer. This is useful for the rather common occurrence of IF–THEN and IF–THEN–ELSE sequences.
3. This strategy is particularly well suited to dealing with loops, or iterations; hence the name loop buffer. If the loop buffer is large enough to contain all the instructions in a loop, then those instructions need to be fetched from memory only once, for the first iteration. For subsequent iterations, all the needed instructions are already in the buffer.

If the buffer contains 256 bytes, and byte addressing is used, then the least significant 8 bits are used to index the buffer. The remaining most significant bits are checked to determine if the branch target lies within the environment captured by the buffer.

3.2.5 Delayed branch

Re-arrange the instructions so that branching occur later than originally specified. This is a software solution.

3.3 Branch prediction

Various techniques can be used to predict whether a branch will be taken. Among the more common are the following:

- Predict never taken.
- Predict always taken.
- Predict by opcode.
- Taken/not taken switch.
- Branch history table

3.3.1 Static Branch Prediction

Predict always taken and **Predict never taken** those two either always predict the jump and fetches the next instruction or never does this respectively.

Predict by opcode branches based on the operation, this can be useful because some operations are more likely to result in a jump than others. For example:

- BNZ (Branch if the result is Not Zero).
- BEZ (Branch if the result equals Zero)

3.3.2 Dynamic Branch Prediction

Here we branch based on the branch history, we can store information regarding branches in a branch-history table so that we more accurately can predict the branch outcome.

3.3.3 Bimodal Prediction

We want to use a 2-bit saturating counter to predict the most common direction, where the first bit indicated the prediction.

Branches that doesn't take the branch will decrement the counter, and branches that takes the branch will increment it. **This makes it possible to tolerate a branch going in the wrong direction one time**.

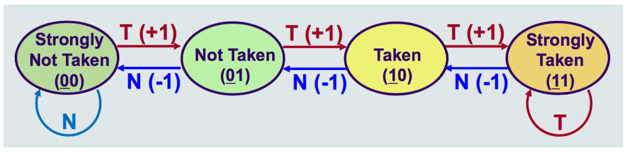


Figure 3 – Bimodal prediction using a 2-bit counter, much like a state diagram.

4 Lecture 4: RISC Computers

4.1 Introduction

Reduced Instruction Set Computer (RISC) represents an important innovation in computer architecture. It is an attempt to produce more computation power by simplifying the instruction set of the CPU.

The opposed trend to RISC it the **Complex Instruction Set Computer** (CISC), or "the regular computers".

Both these architectures have been developed to address problems caused by the **semantic gap**, which is the increasing gap between high level languages (e.g. Java, C++, C#) and low level machine language.

4.1.1 Main features of CISC

- CISC attempts to make machine language (ML) instructions similar to high level languages (HLL)
- Uses a large number of instructions aswell as complex instructions.
- Many and complex addressing modes.
- Microprogramming techniques are used to implement the complicated instructions.
- Memory bottleneck is a major problem, due to complex addressing modes and multiple memory accesses instruction.

4.1.2 Arguments for CISC

- A rich instruction set should simplify the compiler by having instructions who match the HLL instructions.
- Programs are smaller in size and will thus take up less of the memory and smaller execution time due to fewer instructions.
- Program execution efficiency is improved by implementing complex operations in microcode rather than machine code.

4.1.3 Microprogrammed Control

Microprogramming is a technique used to implement the control unit.

- The basic idea is to implement the control unit as a microprogram execution machine (a computer inside a computer).
  – The set of micro-operations occurring at one time defines a microinstruction.
  – A sequence of microinstructions is called a microprogram.

- The execution of a machine instruction becomes the execution of a sequence of micro-instructions.
  – This is similar to that a C++ statement is implemented by a sequence of machine instructions.

Jag tror det är såhär en Olle Roos-dator fungerar med sitt mikrominne, FRÅGA YNGVE
Microcodes are stored in a micromemory which is much faster than a cache, a ROM is often used because we almost never change this memory. Sometimes these memories are called firmwares.

4.1.4 Problems with CISC

As mentioned earlier, with CISC we simplify the compiler, but the hardware goes the opposite way and needs to be more complex. If complex instructions doesn't match the HLL instruction, in which case they may be of little use, this problem is getting bigger and bigger due to that the number of HLL are increasing.

A complex hardware design makes the release time of a processor longer.

4.2 Program execution features

what are programs doing most of the time?
Frequency of machine instructions executed:

- Moving data, 33%
- Conditional branches 20%
- Arithmetic/logic operations 16%
- Others 0.1-10%

The majority of instructions uses simple addressing modes, complex addressing modes are only used by ∼18%.

Operand types:

- 74-80% scalars (integers, reals, characters, etc.).
- the rest (20-26%) are arrays/structures; 90% of them are global variables
- about 80% of the scalars are local variables

Conclusion: The majority of operands are local variables of scalar type, which can be stored in registers.

For HLL statements, most of the time is spent executing CALLs and RETURNs in programs. Even though those only take up 15% it still takes up most of the time due to long executing times for those two.

4.3 RISC characteristics

- Small number of simple instructions (desirable ≤ 100).
- Execution of one instruction per clock cycle.
- Complex operations are executed as a sequence of simple instructions.
- only LOAD and STORE instructions reference data in memory.
- Only a few simple addressing modes are used.
- Instructions are of fixed length and uniform format.
- Large number of registers, this is because the reduced complexity of the processor leaves silicon space on the chip to implement them (opposite of CISC).

4.3.1 Register Windows

Large number of registers is usually very useful. However, if contents of all registers must be saved at every procedure call, more registers mean longer delay. A solution to this problem is to divide the register file into a set of fixed-size windows.
– Each window is assigned to a procedure.
– Windows for adjacent procedures are overlapped to allow parameter passing

4.3.2 Main advantages of RISC

Best support is given by optimizing most used and most time consuming architecture aspects.

- Frequently executed instructions.
- Simple memory reference.
- Procedure call/return.
- Pipeline design.

Consequently, we have:

- High performance for many applications:
- Less design complexity.
- Reduced power consumption.
- reducing design cost and time-to-market (newer technology).

4.3.3 Criticism of RISC

- Operation might take several instructions to accomplish.
- more memory access might be needed.
- Execution speed may be reduced for certain applications.
- It usually leads to longer programs, which needs larger memory space to store.
- It makes it more difficult to program machine codes and assembly programs.

4.4 RISC vs. CISC

Studies have shown that RISC machines often run faster than CISC machines. But there is a problem in deciding what makes RISC better, some say it's not the hardware and instead the compilers for RISC that makes them better. But RISC on the other hand needs more memory because of its simpler instructions.

Most recent CPUs are not strictly RISC or CISC, but more hybrids of both.

In embedded systems RISC is **always** the better choice, due to:

- RISC gives better MIPS/watt ratio.
- RISC reduces power consumption.
- RISC lowers heat dissipation.
- RISC simplifies hardware and its design.

ARM processors with RISC architecture have been widely used in smart phones and computers (e.g., iPad).

5 Lecture 5: Superscalar Architecture (SSA)

Computer designed to improve computation on scalars instructions. A scalar is a variable that can hold only one atomic value at a time, e.g., an integer or a real. A scalar architecture processes one data item at a time – the computers we discussed up till now. Examples of non-scalar variables:

- Arrays – Vector Processor
- Matrices – Graphics Processing Unit (GPU)
- Records

In a superscalar architecture (SSA), several scalar instructions can be initiated simulaneously and executed independently.

5.1 Instruction-level parallelism

Most operations are on scalar quantities, speed up these operations will lead to large performance improvement.

5.1.1 Superpipelining

Superpipelining exploits the fact that many pipeline stages perform tasks that require less than half a clock cycle. Thus, a doubled internal clock speed allows the performance of two tasks in one external clock cycle.

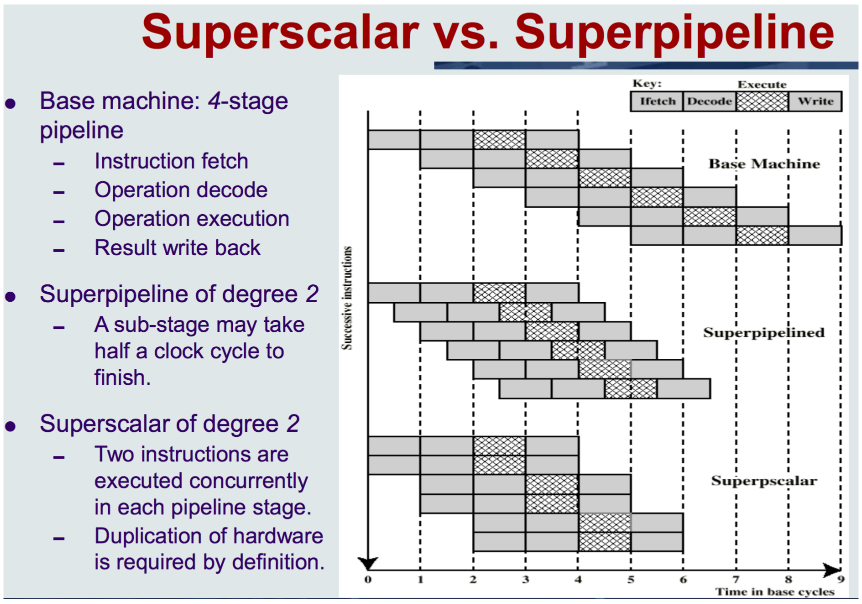
We divide pipelining stages into several sub-stages, and hence increase the number of instructions which are handled by the pipeline at the same time. But note that although several instructions are executing concurrently, only one instruction is in its execution stage at any one time.

A picture would be nice here

1. For example by diving each stage into two sub stages, we will be able (in the ideal situation) to perform each stage at twice the speed.
2. No duplication of hardware is needed.
3. Not all stages can be divided into (equal length) sub stages.
4. Hazards more difficult to resolve.
5. More complex hardware.
6. Interrupt handling and testing will be more complicated.

5.1.2 Difference between Superpipelined and Superscalar Designs

The difference is that Superpipeline can perform several instructions in one clock cycle, whereas Superscalar performs several instructions in parallel

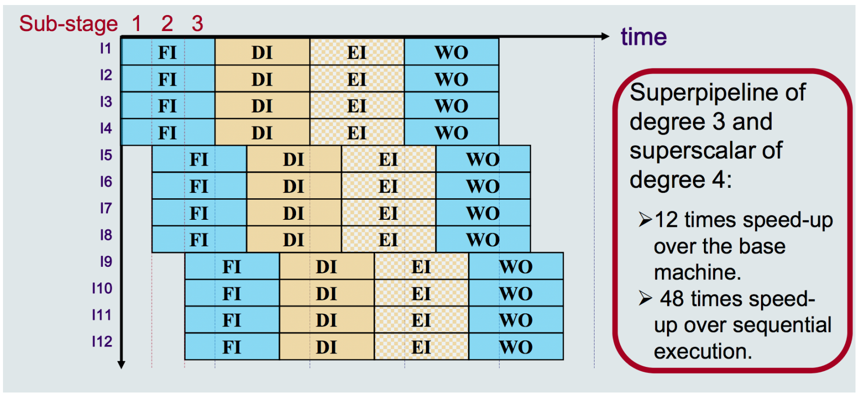


Superscalar vs. Superpipeline

- Base machine: *4*-stage pipeline
  - Instruction fetch
  - Operation decode
  - Operation execution
  - Result write back
- Superpipeline of degree 2
  - A sub-stage may take half a clock cycle to finish.
- Superscalar of degree *2*
  - Two instructions are executed concurrently in each pipeline stage.
  - Duplication of hardware is required by definition.

Figure 4 – Difference between a Superscalar and a Superpipeline architecture

5.1.3 Superscalar Superpipeline Design



Superpipeline of degree 3 and superscalar of degree 4:

- ➢12 times speed-up over the base machine.
- ➢ 48 times speed-up over sequential execution.

Figure 5 – Superscalar Superpipeline architecture

The new trend is to combined the two ideas. In Figure 5 it says that this would give us a 48 times speedup, this is not the case though because of data dependencies.

5.2 Dependency issues

The superscalar approach depends on the ability to execute multiple instructions in parallel. The term instruction-level parallelism refers to the degree to which, on average, the instructions of a program can be executed in parallel. A combination of compiler-based optimization and hardware techniques can be used to maximize instruction-level parallelism.

The main problems are:

- Resource conflicts.
- Control (procedural) dependency.
- Data dependencies.
- True data dependency.
- Output dependency.
- Anti Dependency.

These are very similar to the cases in normal pipelining (data hazards). But the consequences are more severe here because the parallelism are greater and thus larger amount of performance will be lost.

5.2.1 Resource Conflicts

Several instructions compete for the same hardware resource at the same time.

- For instance, two aritmethic instructions need the same floating-point unit for execution.
- similar to structural hazards in pipeline.

They can be solved partly by introducing several hardware units for the same functions.

- e.g. have two floating point units.
- the hardware units can also be pipelined to support several operations at the same time.
- however, memory units **can't be duplicated**.

5.2.2 Procedural Dependency

The instructions following a branch (taken or not taken) have a procedural dependency on the branch and cannot be executed until the branch is executed.

As with regular pipelining, this type of procedural dependency also affects a scalar pipeline. The consequence for a superscalar pipeline is more severe, because a greater magnitude of opportunity is lost with each delay.

If variable-length instructions are used, then another sort of procedural dependency arises. Because the length of any particular instruction is not known, it must be at least partially decoded before the following instruction can be fetched. This prevents the simultaneous fetching required in a superscalar pipeline. This is one of the reasons that superscalar techniques are more readily applicable to a RISC or RISC-like architecture, with its fixed instruction length.

5.2.3 Data Conflicts

These are caused by dependencies between instructions in the program. They are similar to data hazards in regular pipelining, but they cause more problems due to that we much more data dependencies because of parallel execution of many instructions.

To address the problem and to increase the degree of parallel execution, SSA provides a great liberty in the order in which instructions are issued and executed. Therefore, data dependencies have to be considered and dealt with much more carefully.

5.2.4 Window of execution

Because of data dependencies, only some instructions are able to execute in parallel. The processor has to select from a sufficiently large instruction set. **Window of execution** is then the set of instructions that is considered to be able to execute in parallel at a certain moment.

The number of instructions in the window should be as large as possible, however it is limited by:

- Capacity to fetch instructions at a high rate.
- The problem of branches.
- The cost of hardware needed to analyze data dependencies (we don't really care about cost, we want to build cool stuff)

The window of execution can be extended over basic block borders by branch prediction (**speculative execution**).

Speculative execution works like this:

1. Instructions of the predicted path are entered into the window of execution.
2. If the prediction turns out to be correct, we were able to do some of them in advance (which is superduper), and the changes will become permanent and visible (they will **commit**)
3. If not, all changes will undone, and the result will be ignored.

5.2.5 Data Dependencies

All instructions in the window of execution may begin execution, subject to data dependence and resource constraints.

- True data dependency
- Output dependency
- Anti dependency

5.2.6 True Data Dependency

True data dependencies exist when the output of one instruction is required as an input to a subsequent instruction:

```
MUL R4,R3,R1 (R4 := R3 * R1)
 .  .  .
ADD R2,R4,R5 (R2 := R4 + R5)
```

In this example we can fetch and decode the second instruction in parallel, but we have to wait to perform it because its dependant on registers from the first instruction.

They are intrinsic features of a program, and cannot be eliminated by compiler or hardware techniques. The hardware have to detect them, and solve them. The easiest way to do this in this example, is just to stall the first instruction until we can perform the second.

This type of depency is very common, and by increasing the window size we can reduce the impact of them.

5.2.7 Anti Dependency

Also known as write after read, occurs when an instruction requires a value that is later updated. In the code block below, instruction 2 is anti-dependant on instruction 3. The ordering of the instructions cannot be changed, nor can the instructions be executed in parallel.

```
1. B = 3
2. A = B + 1
3. B = 7
```

Anti-dependcies are naming dependencies, renaming of variables could remove the dependency, as seen in the code below.

```
1. B = 3
N. B2 = B
2. A = B2 + 1
3. B = 7
```

5.2.8 Output Dependency

An output dependency, also known as write-after-write (WAW), occurs when the ordering of instructions will affect the final output value of a variable. In the example below, there is an output dependency between instructions 3 and 1 — changing the ordering of instructions in this example will change the final value of A, thus these instructions cannot be executed in parallel.

```
1. B = 3
2. A = B + 1
3. B = 7
```

As with anti-dependencies, output dependencies are name dependencies. That is, they may be removed through renaming of variables, as in the below modification of the above example:

```
1. B2 = 3
2. A = B2 + 1
3. B = 7
```

5.3 Parallel instruction execution

Instruction-level parallelism (ILP) is the average number of instructions in a program that a processor might be able to execute at the same time. It is determined by the number of true dependencies and procedural (control) dependencies in relation to the number of other instructions.

Consider the following two code fragments. The three instructions on the left are independent, and in theory all three could be executed in parallel. In contrast, the three instructions on the right cannot be executed in parallel because the second instruction uses the result of the first, and the third instruction uses the result of the second.

```
1. R1 = R2              1. R3 = R3 + 1
2. R3 = R3 + 1          2. R4 = R4 + R2
3. R4 = R4 + R2         3. R4 = R0
```

Instruction-level parallelism is also determined by operation latency: the time until the result of an instruction is available for use as an operand in a subsequent instruction. The latency determines how much of a delay a data or procedural dependency will cause.

Machine parallelism of a processor, is the ability of the processor to take advantage of the ILP of the program. It it determined by the number of instructions that can be fetched and executed at the same time (the number of parallel pipelines), and by the speed and sophistication of the mechanisms that the processor uses to find independent instructions.

The use of a fixed-length instruction set architecture, as in a RISC, enhances instruction-level parallelism. On the other hand, limited machine parallelism will limit performance no matter what the nature of the program.

To achieve high performance, we need both ILP and machine parallelism. Ideally, we have the same ILP and machine parallelism. However, this is impossible, since the same computer is used for programs with different ILPs.

5.3.1 Division and Decoupling

To increase ILP, we should divide the instruction execution into smaller tasks and decouple them. In particular, we have three important activities:

- **Instruction Issue** – an instruction is initiated and starts execution
- **Instruction issue** – an instruction is initiated and starts execution
- **Instruction commit** – the operation results are written back to the register files or cache (The machine state is changed).

5.3.2 SSA instruction Execution Policies

Instructions can be executed in an order different from the strictly sequential one, with the requirement that **the results must be the same**.

Execution policies usually used:

- In-order issue with in-order completion.
- In-order issue with out-of-order completion.
- Out-of-order issue with out-of-order completion.
- ~~Out-of-order issue with in-order completion~~.

5.3.3 In-Order Issue with In-Order Completion

Instructions are issued in exact program order, and completed in the same order (with parallel issue and completion, of course!).

- An instruction cannot be issued before the previous one has been issued.
- An instruction cannot be completed before the previous one has been completed.

To guarantee in-order completion, an instruction will stall when there is a conflict and when a unit requires more than one cycle to execute.

Example: A processor can issue and decode 2 instructions per cycle, has 3 functional units (2 single-cycle integer units, and 1 two-cycle floating-point unit), and can write back 2 results per cycle.

An instruction sequence with the following characteristics:

- I1 – needs two execute cycles (floating-point)
- I2 –
- I3 –

- I4 – needs the same function unit as I3
- I5 – needs data value produced by I4
- I6 – needs the same function unit as I5

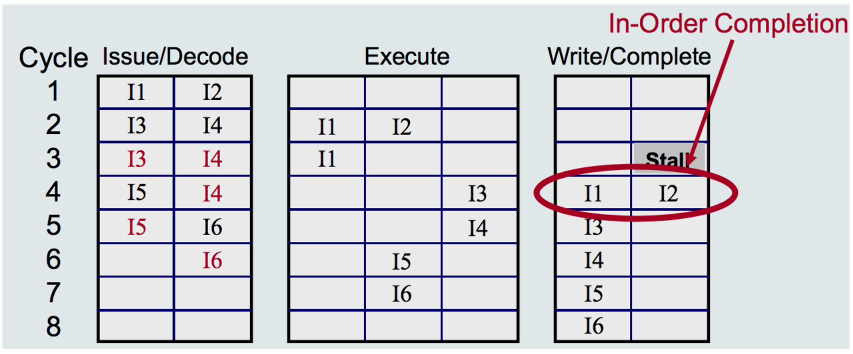


Figure 6 – An example of In-Order Issue with In-Order Completion

The processor detects and handles (by stalling) true data dependencies and resource conflicts, and thus does not rely on compiler-based technique (compatibility consideration)

This is not a very efficient technique, but it simplifies the hardware.

5.3.4 In-Order Issue with Out-of-Order Completion

With out-of-order completion, a later instruction may complete before a previous one. It addresses mainly the issue of long-latency operations, such as division.

With out-of-order completion, any number of instructions may be in the execution stage at any one time, up to the maximum degree of machine parallelism across all functional units. Instruction issuing is stalled by a resource conflict, a data dependency, or a procedural dependency.

Example:

- I1 – needs two execute cycles
- I2 –
- I3 –
- I4 – conflict with I3
- I5 – depending on I4
- I6 – conflict with I5

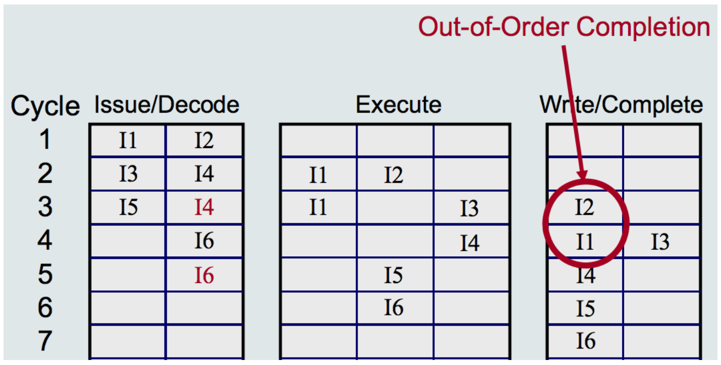


Figure 7 – An example of In-Order Issue with Out-Order Completion

Now instruction I2 can write back, and doesn't have to wait for I1.
tror inte jag fattar det här rätt, kanske borde skriva om detta.
det står bättre i kursboken (runt sida 580), kanske borde skriva om det istället

5.3.5 Out-Order Issue with Out-of-Order Completion

With **in-order issue**, no new instruction can be issued when the processor has detected a conflict, and is stalled until after the conflict has been resolved. The processor is not allowed to look ahead for further instructions, which could be executed in parallel with the current ones.

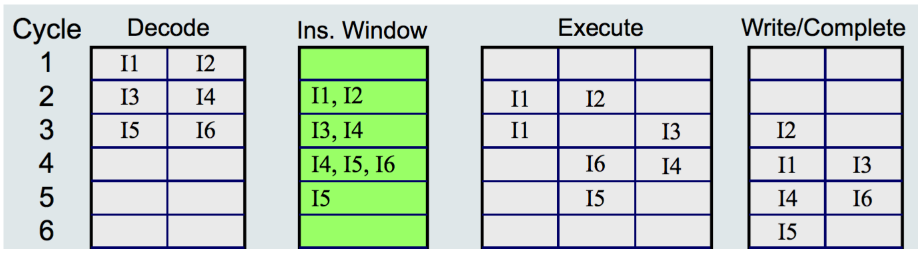
With **out-of-order issue** we can continue to fetch new instructions even though we have to stall in a later stage.

We can do this by decouple the decode and execution stages of the pipeline. We then use a buffer, called the **instruction window** which we put the instructions in when we have fetched and decoded them, note that this can only be done if this buffer if not full.

When an functional unit becomes free, an instruction from the instruction window may be issued to the execution stage. Any instruction may be issued as long as it is of the right type for the free unit, and it doesn't inflict any conflict.

Example:

- I1 – needs two execute cycles
- I2 –
- I3 –
- I4 – conflict with I3
- I5 – depending on I4
- I6 – conflict with I5



Cycle	Decode		Ins. Window	Execute			Write/Complete	
1	I1	I2						
2	I3	I4	I1, I2	I1	I2			
3	I5	I6	I3, I4	I1		I3	I2	
4			I4, I5, I6		I6	I4	I1	I3
5			I5		I5		I4	I6
6							I5	

Figure 8 – An example of Out-Order Issue with Out-Order Completion

This works as a look-ahead capability for the processor. Instructions are issued from the instruction window with little regard for their original program order. As before, the only constraint is that the program execution behaves correctly.

det står mer om detta i kursboken också sida 583-584

6 Lecture 6: VLIW Processors

Good things about SSA:

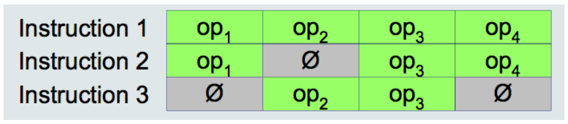
- The difference from Superscalar is that in that architecture the hardware solves everything for us.
- It detects potential parallelism between instructions.
- It tries to issue as many instructions as possible in parallel.
- It uses register renaming to increase parallelism.
- if improvements are made on the design we don't need to change the programs.
- Old programs can benefit from the additional machine parallelism, since the new hardware will simply issue instructions in a more efficient way.

Problems with SSA:

- The down side though is that the architecture is very complex.
- A lot of hardware is needed for run-time detection of parallelism.
- It consumes a lot of power.
- There is, therefore a limit in how far we can go with this technique.
- The instruction window for execution is limited in size, this limits the capacity to detect large number of parallel instructions.

6.1 Very Long Instruction Word Processors

In VLIW architecture, Several operations that can be executed in parallel are placed in a single instruction word as can be seen in figure 9.



Instruction 1	op_1	op_2	op_3	op_4
Instruction 2	op_1	Ø	op_3	op_4
Instruction 3	Ø	op_2	op_3	Ø

Figure 9 – VLIW instruction word

VLIW rely on compile-time detecton of parallelism. The compiler analyzes the program and detects operations to be executed in parallel.

After one instruction has been fetched, all the corresponding operations are issued in paralell. The instruction window limitation disappears: the compiler can potentially analyze the whole program to detect parallel operations.

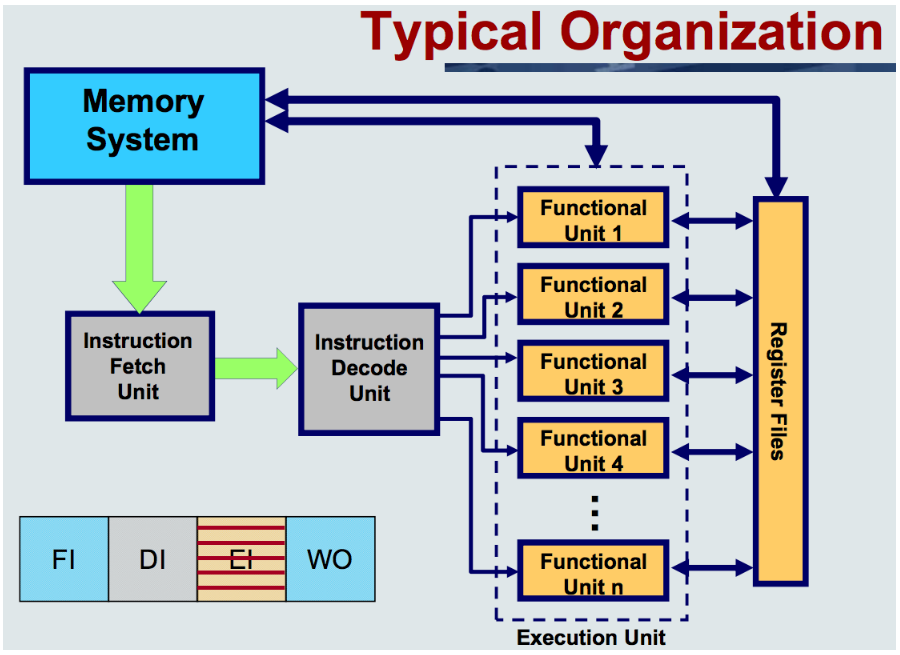


Figure 10 – Typical VLIW organization

6.1.1 Explicit Parallelism

Instruction parallelism scheduled at compile time. It is included within the machine instructions explicitly. This means that the hardware is very much simplified, and that functional units can be added withouth need of additional sophisticated hardware to detect parallelism, as in SSA.

An EPIC (Explicitly Parallel Instruction Computing) processor uses this information to perform parallel execution.

As it is the Compiler that determines the parallel operations, this is done off-line and has much more time and is therefore not as time critical as SSA, as this is done at run-time by the hardware in that case. Good compilers can detect parallelism based on global analysis of the whole program.

6.1.2 Main issues

But there is also some problems with VLIW aswell.

- Needs a large number of registers.
- Large data transport capacity is needed between Fus and the register files and between register files and memory.
- High bandwidth between instruction cache and fetch unit is also needed due to long iunstructions.

- Every code block is not optional for the design, which leads to that some FUs might be unused a lot of the time.

6.1.3 Software issues

If a new version of the processor introduces additional FUs, the number of operations to execute in parallel is increased. Therefore, the instruction word changes, and old binary code cannot be run on the new processor

We might not have sufficient parallelism in the program to utilize the large degree of machine parallelism.

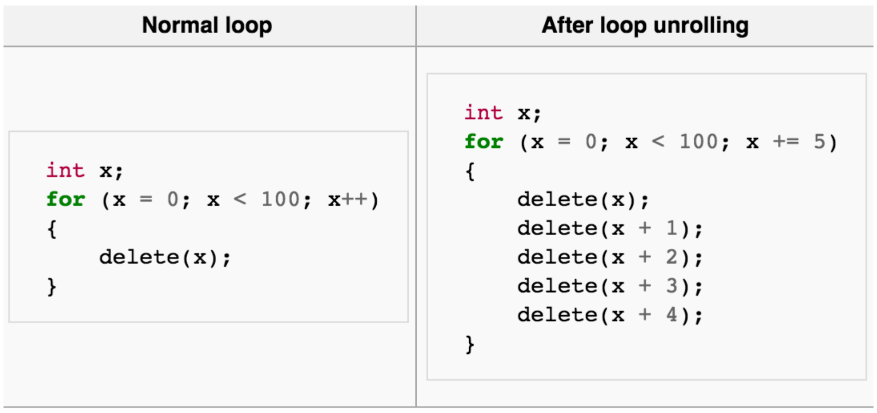
- Hardware resources will be wasted in this case.
- A lot of memory space will also be wasted.
- A technique to address this problem is loop unrolling, which can be performed by a compiler.

6.2 Loop unrolling

Loop unrolling: a technique used in compilers in order to increase the potential of parallelism in a program. It supports more efficient code generation for processors with instruction level parallelism.

What this means is that we will try to re-write loops to eliminate instructions that controls the loop.

An example is given in figure 11.



Normal loop

```
int x;
for (x = 0; x < 100; x++)
{
    delete(x);
}
```

After loop unrolling

```
int x;
for (x = 0; x < 100; x += 5)
{
    delete(x);
    delete(x + 1);
    delete(x + 2);
    delete(x + 3);
    delete(x + 4);
}
```

Figure 11 – A simple example of loop unrolling.

Before using loop unrolling, we have 100 iterations, after we only need 20 iterations. This means that only 20% of the jumps and conditional branches that were taken before is taken now. What this really does, is that it makes it possible for us to pack the instructions tighter and execute the loop in far less cycles.

But there is also a limit in have many iterations we can unroll, depending on the architecture.

Loop unrolling also increases the amount of:

- Memory space needed to store the instructions.
- The amount of register needed to store data for operations that are performed in parallel.

6.3 IA-64 architecture

The key features of IA-64 are:

- **Large number of registers**: The IA-64 instruction format assumes the use of 256 registers.
- **Multiple execution units**.

Four types of execution unit are defined in the IA-64 architecture:

- **I-unit**:Forintegerarithmetic,shift-and-add,logical,compare,andintegermultimedia instructions
- **M-unit**: Load and store between register and memory plus some integer ALU operations
- **B-unit**: Branch instructions
- **F-unit:** Floating-point instructions

The basic idea of the architecture can be seen in figure **??**

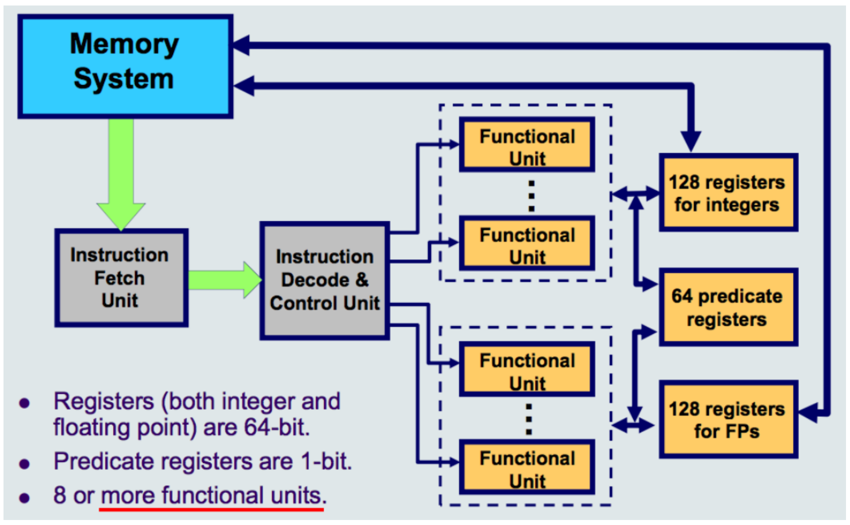

Memory System
Instruction Fetch Unit
Instruction Decode & Control Unit
Functional Unit
Functional Unit
Functional Unit
Functional Unit
128 registers for integers
64 predicate registers
128 registers for FPs
Registers (both integer and floating point) are 64-bit.
Predicate registers are 1-bit.
8 or more functional units.

Figure 12 – IA-64 Architecture

6.3.1 Instruction format

In figure 13 we can see how the instruction format are built.

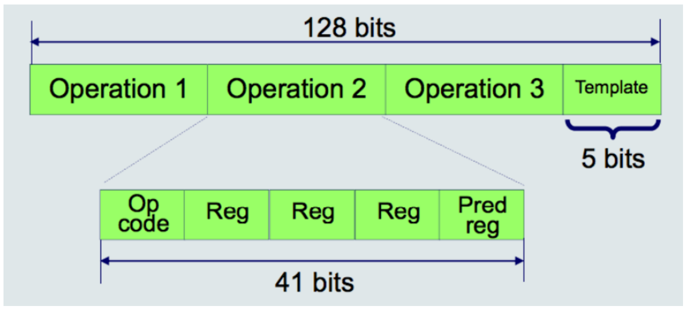


Figure 13 – Instruction format forIA-64.

Three operations are specified in an instruction word, but it's not sure that we can execute them in parallel! This is what we use the template for.

The template tells what instructions in the instruction word that can be executed in parallel, it also connects **neighboring instructions**. Therefore, operations from different instructions can also be executed in parallel

The positive things

6.3.2 Predicated execution

- Any operation can refer to a predicate register. <Pi> operation where i is the number of a predicate register (between 0 and 63)
- This means that an operation is to be committed (the results made permanent) only when the respective predicate is true (i.e., the predicate register gets value 1).
- If the predicate value is known when the operation is issued, the operation is executed only if this value is true.
- If the predicate is not known at that moment, the operation will also be started. If the predicate turns out to be false, the operation is discarded.
- If no predicate register is mentioned, the operation is executed and committed unconditionally

6.3.3 Branch Predication

We can first note that this is **not** the same as **Branch Prediction**.

It is an compiler technique that lets both branches of a conditional branch to be executed in parallel, to increase the amount of parallel instructions.

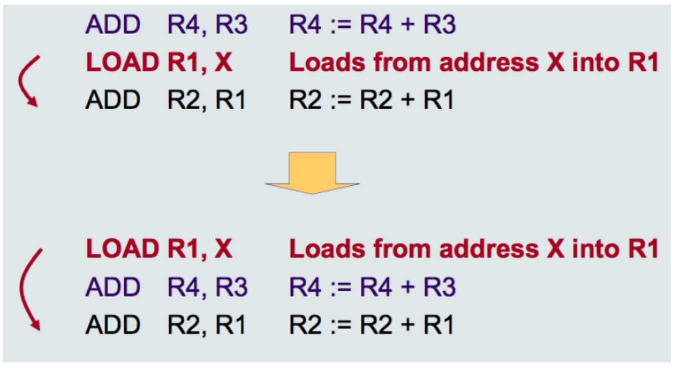
This is kind of like guessing both answer answers in a yes/no questions. The branch that was correct will then commmit its result and we will just ignore/undo the results of the branch that was incorrect.

This technique of course need duplicated hardware (given by the IA-64 architecture), and it will mean that some of the hardware will be wasted instead of doing something useful.

The upside is that we won't loose any time with branch misses.

6.3.4 Placement of Loading

An operation that loads from memory should be placed, so that memory latency is avoided. An example of this can be seen in figure 14



```
ADD   R4, R3     R4 := R4 + R3
LOAD R1, X       Loads from address X into R1
ADD   R2, R1     R2 := R2 + R1

LOAD R1, X       Loads from address X into R1
ADD   R4, R3     R4 := R4 + R3
ADD   R2, R1     R2 := R2 + R1
```

Figure 14 – We can avoid memory latency by loading from the memory in good time before we need the memory.

6.3.5 Speculative Loading

Speculative loading tries to reduce latencies generated by load instructions. It allows load instructions to be moved across branch boundaries (15), and so page exceptions will be handled only if really needed.

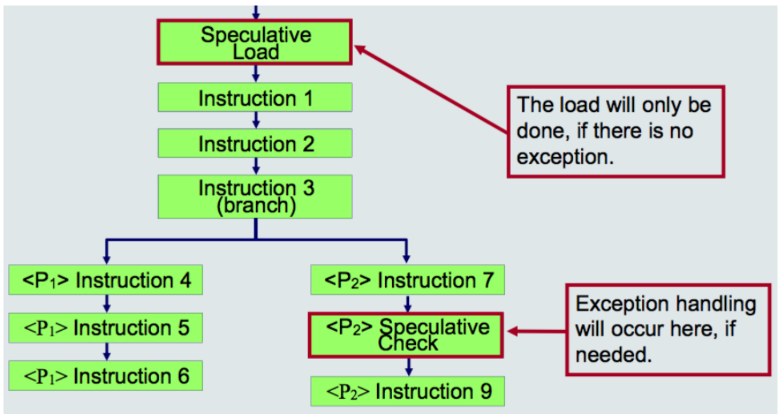


Figure 15 – Speculative loading.

7 Lecture 7: Parallel Processing

7.0.1 Why Parallel Processing?

7.0.2 Parallel Computer

7.0.3 Parallel Program

Allt det här känns som repetition, borde jag skippa?

7.1 Architecture classification

7.1.1 Flynn's Classification of Architectures

- Single instruction, single data stream - **SISD**
- Single instruction, multiple data stream - **SIMD**
- Multiple instruction, single data stream - **MISD**
- Multiple instruction, multiple data stream- **MIMD**

7.1.2 SISD

The regular computers who use a single processor, a single instruction stream, and stores data in a single memory.

7.1.3 SIMD

Computers with multiple processing elements that perform the same operation on multiple data points simultaneously. There are simultaneous (parallel) computations, but only a single process (instruction) at a given moment.

Array and vector processors are the most common examples of SIMD machines.

7.1.4 MISD

Computers where many functional units perform different operations on the same data.

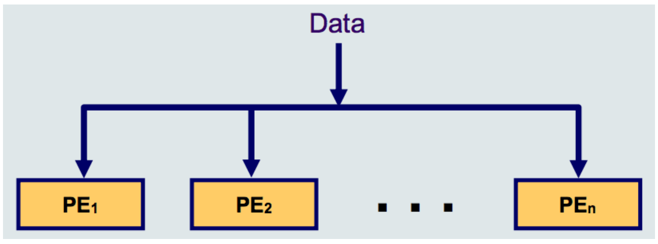


Figure 16 – MISD architecture

7.1.5 MIMD

A set of processors that simultaneously perform different instructions sequences, on different sets of data.

There is also different classes of MIMD computers:

- Shared memory (tightly coupled).
  - Symmetric multiprocessor (SMP).
  - Non-uniform memory access (NUMA).
- Distributed memory (loosely coupled) = Clusters.

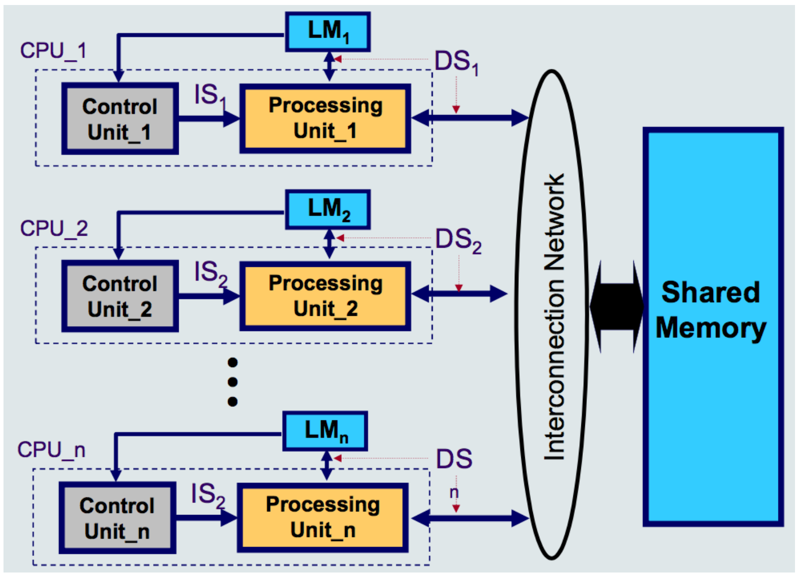


Figure 17 – MIMD architecture

7.2 Performance evaluation

The peak rate of a computer is the maximum value that the computer can work at, this is something that vendors use to sell computers and not something that is really useful for us.

Speedup: measures the gain we get by using a parallel computer, over a sequential one, to run a given application

$$S = \frac{T_s}{T_p}$$

T_s : execution time needed with the sequential computer.
T_p : execution time needed with the parallel computer.

Efficiency: to relate speedup to the number of processors used, it provides therefore a measure of the efficiency with which the processors are used.

$$E = \frac{S}{P}$$

S: speedup.
P: number of processors.
For the ideal situation, in theory: $S = P$; which means $E = 1$.

Practically the ideal efficiency of 1 cannot be achieved!

7.2.1 Amdahls Law

Gives the theoretical speedup in latency of the execution of a task at fixed workload that can be expected of a system whose resources are improved.

$$S_{latency}(s) = \frac{1}{1 - p + \frac{p}{s}}$$

where

- $S_{latency}$ is the theoretical speedup in latency of the execution of the whole task;
- s is the speedup in latency of the execution of the part of the task that benefits from the improvement of the resources of the system;
- p is the percentage of the execution time of the whole task concerning the part that benefits from the improvement of the resources of the system before the improvement.

Furthermore:
$$\begin{cases} S_{latency}(s) \leq \frac{1}{1-p} \\ \lim_{s\to\infty} = \frac{1}{1-p} \end{cases}$$

show that the theoretical speedup of the execution of the whole task increases with the improvement of the resources of the system and that regardless the magnitude of the improvement, the theoretical speedup is always limited by the part of the task that cannot benefit from the improvement.

Amdahl's law is often used in parallel computing to predict the theoretical speedup when using multiple processors. For example, if a program needs 20 hours using a single processor core, and a particular part of the program which takes one hour to execute cannot be parallelized, while the remaining 19 hours (p = 0.95) of execution time can be parallelized, then regardless of how many processors are devoted to a parallelized execution of this program, the minimum execution time cannot be less than that critical one hour. Hence, the theoretical speedup is limited to at most 20 times (1/(1 − p) = 20). For this reason parallel computing is relevant only for a low number of processors and very parallelizable programs. This can be seen in figure 18

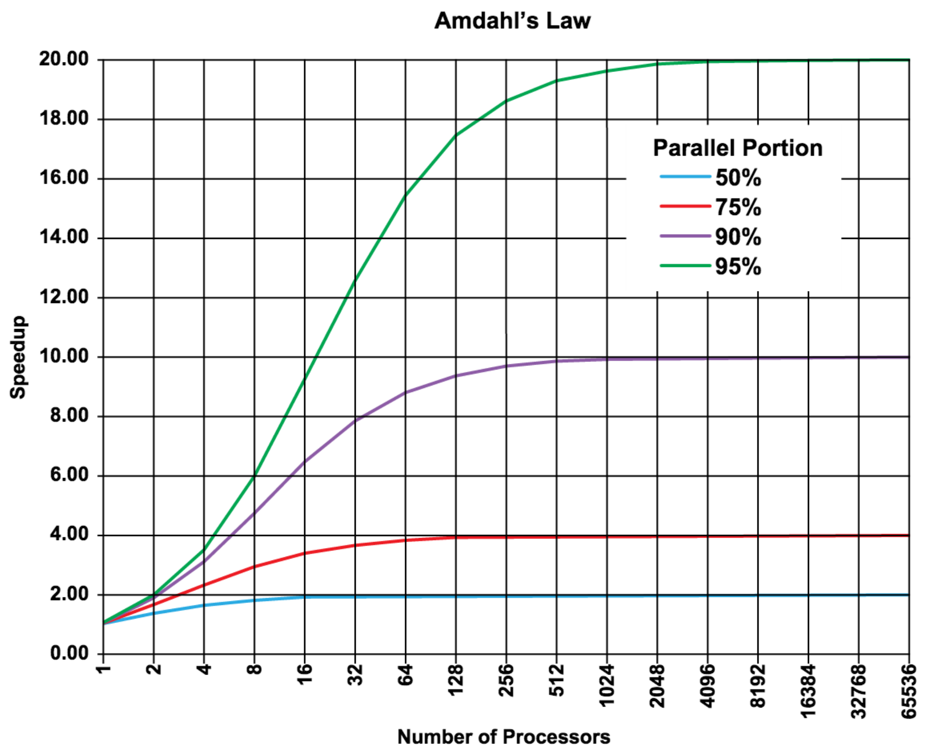


Figure 18 – Amdahls Law

skipped other factors that limit speedup and impact of communications

7.3 Interconnection network

Interconnecting Networks (IN) is a key component of parallel computers and has a decisive influence on

- overall performance
- total cost of the architecture

The traffic in an IN consists of:

- data transfer
- transfer of commands and requests (control information).

The key parameters of an IN are:

- total bandwidth: transferred bits/second.
- implementation cost.

7.3.1 Single Bus

These are simple cheap and relatively flexible. Only one communication is allowed at one time, which means the bandwidth is shared by all nodes.

Its performance is relatively poor, in order to get a good performance the number of nodes needs to be limited (16-20), and we can use several buses instead.

7.3.2 Completely Connected Network

Here every is connected to each other, which means that communication can be performed in parallel between any pair of nodes. Performance is high but also the construction cost, which will increase rapidly by every added node.

7.3.3 Crossbar Network

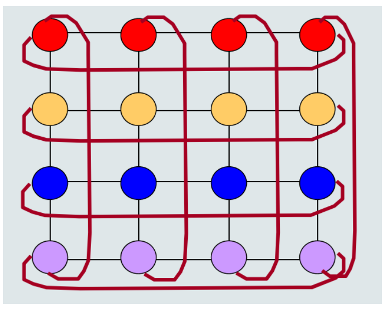
A dynamic network: the interconnection topology can be modified by configurating the switches. It is completely connected: any node can be directly connected to any other
borde det inte stå att: kan vara completely connected?

7.3.4 Mesh Network

- Cheaper than completely connected networks, while giving relatively good performance.
- In order to transmit data between two nodes, routing through intermediate nodes is needed (maximum 2(n-1) intermediates for an n^2 mesh).
- It is possible to provide wrap-around connections: Torus.
- Three dimensional meshes have also been implemented.

7.3.5 Hypercube Network

2^n nodes are arranged in an n-dimensional cube. Each node is connected to n neighbors.


(a) Regular mesh network

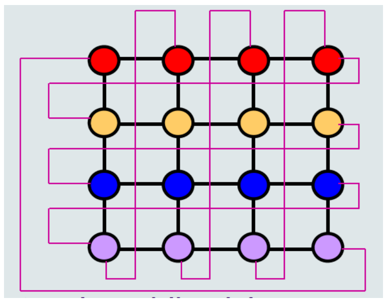

(b) Torus network

Figure 19 – Mesh Network

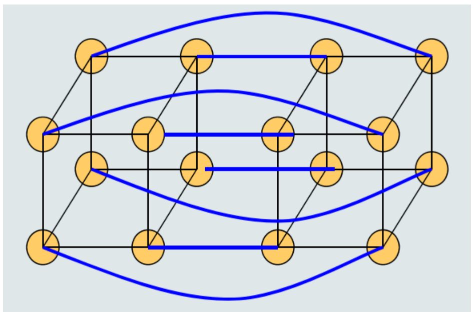


Figure 20 – Hypercube Network

8 Lecture 8: SIMD Architectures

8.1 Vector Processors

– array processors
– vector processors
– data parallelism
Typical SISD processors who behave like SIMD processors

8.1.1 Instruction-level parallelism

8.1.2 Thread-level parallelism

8.1.3 Data parallelism

8.1.4 Vector Processors

8.2 Array Processors

Typical SIMD processors

8.3 Dedicated Memory Organization

8.4 Global Memory Organization

8.5 Cray supercomputers

8.5.1 Cray X1

8.6 Multimedia extensions

8.6.1 sub-word execution

8.6.2 Packed Data Types

8.6.3 SIMD Arithmetic Examples

9 Lecture 9: MIMD Architectures

A set of general purpose processors connected together
In contrast to a SIMD computer, a MIMD computer can execute different programs on different processors.

– Works asynchronously, and don't have to synchronize with each other.
– At any time, different processors may be executing different instructions on different pieces of data.
– They can be built from commodity (off-the-shelf) microprocessors with relatively little effort.
– They are also highly scalable, provided that an appropiate memory organization is used.
– Most current parallel computer are built based on the MIMD architecture.

9.0.1 SIMD vs. MIMD

9.0.2 MIMD Processor Classification

9.0.3 MIMD with Shared Memory

9.0.4 MIMD with Distributed Memory

9.0.5 Shared-Address-Space Platforms

9.0.6 Multi-Computer Systems

9.0.7 MIM Design Issues

9.1 Symmetric multiprocessors (SMP)

9.1.1 SMP Advantages

9.1.2 SMP based on Shared Bus

9.1.3 Multi-Port Memory SMP

9.1.4 Operation System Issues

9.1.5 IBM S/390

9.2 NUMA Architecture

9.2.1 Memory Access Approaches

9.3 Clusters

9.3.1 Losely Coupled MIMD - Clusters

9.3.2 Clusters benefits

9.3.3 Clusters Configurations

9.3.4 IBM Blue Gene Supercomputer

9.3.5 Parallelizing Computation

9.3.6 Google Applications

10 Lecture 10: Cache Coherence

When using cache in multiprocessor systems, different processor may access the same data in memory. This gives us a problem with that we have multiple copies of the same data in different caches. This is known as the **cache coherence** problem.

10.1 Introduction

How can we assure that the other processors get the update in another processors cache, and that we don't overwrite some memory that is critical? We must ensure that **WRITE** operations is carefully coordinated.

10.1.1 Write Through

Write through: All write operations are made to main memory as well as to the cache, ensuring that main memory is always valid.

But there are some problems with this:

- Can slow down the main memory access since we need to write a lot, and this takes time, but since the write percentage is only about 15% its okay.
- Inconsistency can occur unless other caches monitor the memory traffic or receive some direct notification of the update.
- Multiple CPUs must therefore monitor main memory traffic to keep local cache up to date.
  - This may lead to a lots of traffic and monitoring activities.

10.1.2 Write Back

Write back: Write operations are usually made only to the cache. Main memory is only updated when the corresponding cache line is flushed from the cache.

- We update a bit when the cache is updated.
- If we need to replace a bit, we look at the bit to see if the cache has been updated and it needs to be saved to the main memory.

It is clear that a write-back policy can result in inconsistency. If two caches contain the same line, and the line is updated in one cache, the other cache will unknowingly have an invalid value. Subsequent reads to that invalid line produce invalid results. Therefore we need additional techniques.

10.1.3 Software Solutions

Here we deal with the problem during **compile time**, this gives us time to determine which data item that may become unsafe for caching, (e.g. global parameters are typical).

We can either mark these data entries so that they are not cached. This is too conservative, because a shared data structure may be exclusively used during some periods and may be effectively read-only during other periods. It is only during periods when at least one process may update the variable and at least one other process may access the variable that cache coherence is an issue.

Alternatively we can determine the unsafe periods and insert code to enforce cache coherence.

Pros with software solutions:

- Avoid the need for additional hardware circuitry and logic by relying on the compiler and operating system to deal with the problem.

Cons with software solutions:

- We may get inefficient cache utilization and we might therefore get low performance of the memory system.

10.1.4 Hardware Solutions

Hardware-based solutions are generally referred to as cache coherence protocols. These solutions provide dynamic recognition at run time of potential inconsistency conditions. Because the problem is only dealt with when it actually arises, there is more effective use of caches, leading to improved performance over a software approach. In addition, these approaches are transparent to the programmer and the compiler, reducing the software development burden.

In general, hardware schemes can be divided into two categories: **directory protocols** and **snoopy protocols**.

10.2 Directory protocols

Here we have a collections that maintain information about where copies of lines reside. Typically a centralized controller that is part of the main memory controller and a directory stored in the main memory. This works as following:

- When an individual cache controller makes a request, the centralized controller checks and issues necessary commands
- Instead of reading data from the main memory we instead may fetch data from another cache where the data have been updated.
- It keeps the state information up to date. Therefore, every local action that can affect the global state of a line must be reported to the central controller.

- Grants exclusive memory access to processor, by forcing all other processor to invalidate its copy.
- If a processor tries to read a line that is exclusively granted to another processor, it will send a miss notification to the controller. The controller then issues a command to the processor holding that line that requires the processor to do a write back to main memory. The line may now be shared for reading by the original processor and the requesting processor.

Directory schemes suffer from the drawbacks of a central bottleneck and the overhead of communication between the various cache controllers and the central controller. However, they are effective in large-scale systems that involve multiple buses or some other complex interconnection scheme.

10.2.1 Cache Coherence Operations

byt ut denna sektion mot ett exempel kanske.

10.3 Snoopy protocols

Snoopy protocols distribute the responsibility for maintaining cache coherence among all of the cache controllers in a multiprocessor. A cache must recognize when a line that it holds is shared with other caches. When an update action is performed on a shared cache line, it must be announced to all other caches by a broadcast mechanism. Each cache controller is able to "snoop" on the network to observe these broadcasted notifications, and react accordingly.

Snoopy protocols are ideally suited to a bus-based multiprocessor, because the shared bus provides a simple means for broadcasting and snooping. But this also means that we get increased bus traffic.

Two basic approaches to the snoopy protocol have been explored, **write invalidate** and **write update** (or write broadcast). Neither of these two approaches is superior to the other under all circum- stances. Performance depends on the number of local caches and the pattern of memory reads and writes. Some systems implement adaptive protocols that employ both write-invalidate and write-update mechanisms.

10.3.1 Write Invalidate SP

With a write-invalidate protocol, there can be multiple readers but only one writer at a time. Initially, a line may be shared among several caches for reading purposes. When one of the caches wants to perform a write to the line, it first issues a notice that invalidates that line in the other caches, making the line exclusive to the writing cache. Once the line is exclusive, the owning processor can make cheap local writes until some other processor requires the same line.

The write-invalidate approach is the most widely used in commercial multiprocessor systems, such as the Pentium 4 and PowerPC. It marks the state of every cache line (using two extra bits in the cache tag) as modified, exclusive, shared, or invalid. For this reason, the write-invalidate protocol is called MESI.

10.3.2 Write Update SP

With a write-update protocol, there can be multiple writers as well as multiple readers. When a processor wishes to update a shared line, the word to be updated is distributed to all others, and caches containing that line can update it.

But it may generate many unnecessary updates if a processor:

- Reads a value without ever needing it.
- Updates a value several times before it is read by other processors.

10.3.3 MESI State Transition Diagram

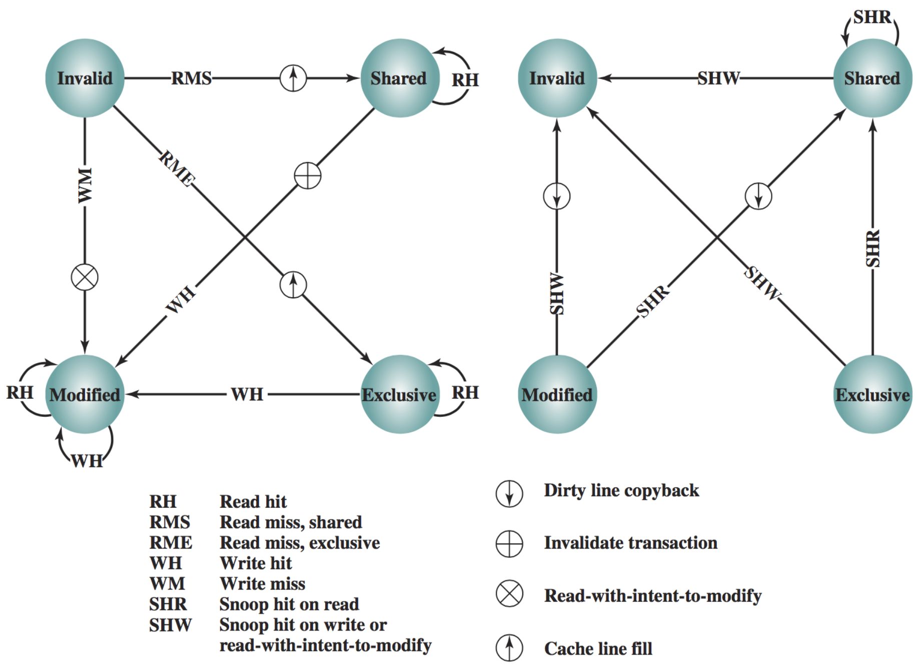


Figure 21 – MESI State Diagram.

UNDERSTAND THIS

10.3.4 Invalidate vs. Update Protocols

- An update protocol may generate many unnecessary cache updates, so if two processors make interleaved reads and updates to a variable, an update protocol is better.

- Both protocols suffer from false sharing overheads, which means that when two words are not shared but they lie within the same cache line.
- Most modern machines use invalidate protocols, since we have usually the case of one writer and many readers.

10.3.5 Directory vs. Snoopy Schemes

- Snoopy caches
  - Each coherence operation is sent to all processors.
  - It generates large traffic, which is an inherent limitation.
  - Easy to implement on a bus-based system.
  - Not feasible for machines with memory distributed across a large number of sub-systems.
- Directory caches
  - The need for a broadcast media is replaced by the directory.
  - The additional information stored in the directory may add significant overhead.
  - The underlying network must be able to carry all the coherence requests.
  - The directory is a point of contention, therefore, distributed directory schemes are often used.

10.4 L1-L2 consistence

When we have several layers of cache we can't use snoopy protocols since the L1 cache won't be connected to a bus (L2 is connected). The solution is to extend cache coherence protocols to L1 cache, which means that the L1 line must keep track of the corresponding state of the L2 cache, and L1 should write-through to L2. If L1 uses write-back, we ge a more complex solution.

But this requires the following:

- L1 must be a subset of L2
- The associativity of L2 must be equal or greater than that of L1.

10.4.1 Alpha-Server 4100

- Four-processor shared-memory symmetric multiprocessor system.
- Each processor has a three-level cache hierarchy:
  - L1 consists of two direct-mapped on-chip caches, onefor instruction and one for data.

    * Write-through to L2 with a write buffer.

- L2 is an on-chip three-way set associative cache with write-back to L3.
- L3 is a off-chip direct-mapped cache with write-back to main memory.

11 Lecture 11: Multi-Core and GPU

11.1 Multi-Core Computers

Integration of several cores on a single chip, this is a cheap and simple solution to gain more parellel computing. This has also been called **Chip Multiprocessor**. A **MIMD processor** have different cores operating on different sets of data. In a **shared memory multiprocessor**, all cores share the same memory via some cache system.

Pros:

- High power consumption in relation to performance gained (2-3% power increase when 1% performance increase).
- Power generates heat and cooling is expensive.
- Instruction Level Parallelism only
- General trend in computer architecture -¿ shift towards more parallelism.

Cons:

- Design time and complexity increased due to complex methods to increase ILP.
- Many new applications are multithreaded, e.g. Multimedia applications.
- Much faster cache coherency circuits, in a single chip.
- Smaller in physical size than SMP.

The idea of multicores processor can be seen in figure 22.

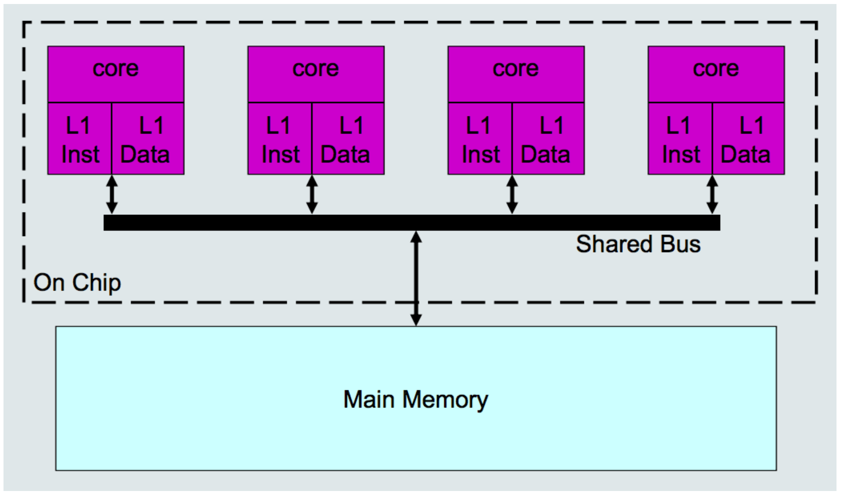


Figure 22 – A simplified model of a multicore processor.

There are also lots of other ideas of how this can be implemented, some of these can be seen in figure 23

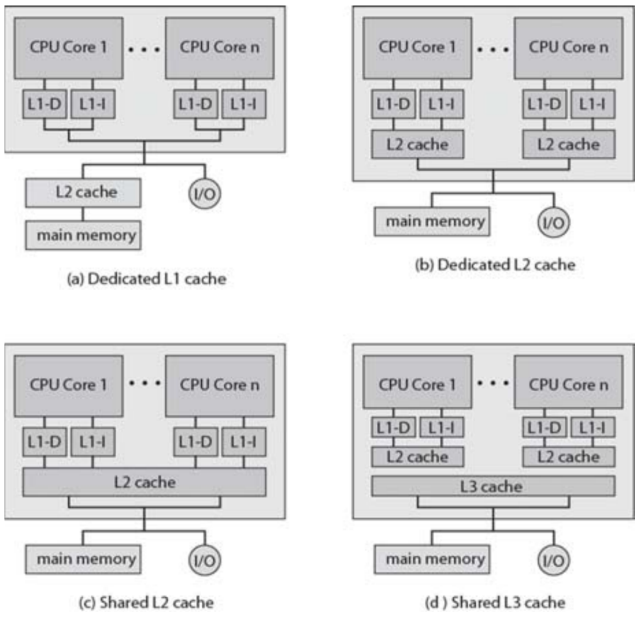


Figure 23 – Some different ideas of how to implement multicore processors.

11.1.1 Intel Core i7

This is a processor with the concept of both several cores and several threads (multithreads).

A simple schedule of the architecture can be seen in figure 24

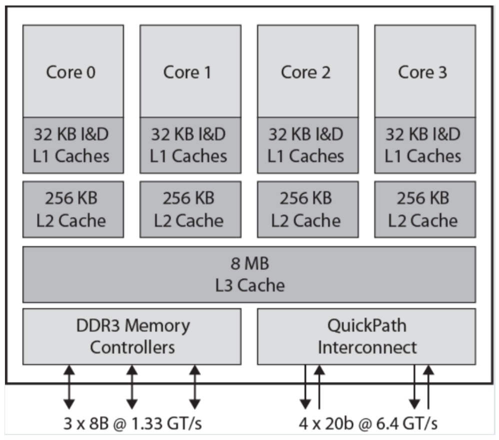


Figure 24 – Intel i7 architecture.

- Four or six identical x86 processors.
- Each with their own L1 & L2 cache.
- Shared L3 cache.
- **Front side bus** (FSB) have been replaced by **QuickPath Interconnect** QPI to increase the bandwidth between cores.
- Up to 4 GHz clock frequency.

11.1.2 Intel Polaris

This processor is developed in cooperation bewtween Linköping University and Intel. It has 80 cores running on one chip, it runs at tFLOPS (10^{12}) performance, making it the first ever chip to perform at that level.

- Its peak power efficiency is 19.4 gFLOPS/Watt (almost 10X better than supercomputers).
- Workload-aware power management
  - The individual computing engines and data routers in each core can be activated or put to sleep based on the performance required by the applications.
- Mesh network-on-a-chip.
  - The cores are connected in a 2D mesh network that implement message-passing.
- Frequency target at 5 GHz.
- Peak of 1.01 Teraflops at 62 watts.
- Short design time (regularity).
  - The tiled-design approach allows designers to use smaller cores that can easily be repeated across the chip.

11.2 Multithreading

11.2.1 Thread-Level Parallelism (TLP)

Parallelism on a larger level than instruction level parallelism ILP. Instruction stream is divided into smaller streams (threads) that will be executed in parallel. Each thread has its own instructions and data. It is also more cost-effective than ILP.

This technique is very suitable for multi-core arhcitectures, as they can use multiple instruction streams to improve the throughput of computers that run several programs.

11.2.2 Multithreading Approaches

jag förstår inte riktigt det här? finns massvis med förklarande bilder som dessvärre inte förklarar så bra i slides

- Interleaved (fine-grained)
  - Processor deals with several thread contexts at a time.
  - Switching thread at each clock cycle (hardware support needed).
  - If a thread is blocked, it is skipped.
  - Hide latency of both short and long pipeline stalls.
- Blocked (coarse-grained)
  - Thread executed until an event causes delay (e.g., cache miss).
  - Relieves the need to have very fast thread switching.
  - No slow down for ready-to-go threads.
- Simultaneous (SMT)
  - Used in **superscalar architectures**.
  - Instructions simultaneously issued from multiple threads to execution units of a superscalar processor.
- Chip multiprocessing
  - Each processor handles separate threads

11.2.3 Interleaved (fine-grained)

The purpose is to remove all data dependency stalls from the execution pipeline. Since one thread is relatively independent from other threads, there's less chance of one instruction in one pipelining stage needing an output from an older instruction in the pipeline.

11.2.4 Blocked (coarsed-grained)

The simplest type of multithreading occurs when one thread runs until it is blocked by an event that normally would create a long-latency stall. Such a stall might be a cache miss that has to access off-chip memory, which might take hundreds of CPU cycles for the data to return. Instead of waiting for the stall to resolve, a threaded processor would switch execution to another thread that was ready to run. Only when the data for the previous thread had arrived, would the previous thread be placed back on the list of ready-to-run threads.

11.2.5 Simultaneous (SMT)

The most advanced type of multithreading applies to superscalar processors. Whereas a normal superscalar processor issues multiple instructions from a single thread every CPU cycle, in simultaneous multithreading (SMT) a superscalar processor can issue instructions from multiple threads every CPU cycle. Recognizing that any single thread has a limited amount of instruction-level parallelism, this type of multithreading tries to exploit parallelism available across multiple threads to decrease the waste associated with unused issue slots.

11.2.6 Scalar Processor Approaches

11.2.7 Superscalar Approaches

11.2.8 SMT and Chip Multiprocessing

11.2.9 Multithreading Paradigms

11.3 Graphic Processing Unit (GPU)

Normal CPUs are designed to handle huge data volumes in serial, one operation at a time (control flow stream).

GPUs instead are designed to handle huge data volumes in parallel in a data stream based flow

When we want to compute a frame for HDTV, there are $2.1 * 10^(6)$ Pixels that needs to be computed. All of the pixels are independent of each other and we have a lot of spatial locality (regular memory access). This means that we could achieve lots of speedups if we increase the hardware.

Each pixel can take a long time to process, as long as we make sure that we process many of them at the same time. What we need for this is a lot of simple parallel processors running at low clock speed. This means that we focus on a high throughput instead of a low latency.

CPU:

- Optimized for low latency.
- A few massive cores, running at very high speeds.
- Lots of hardware for control, few ALUs.

GPU:

- Optimized for high throughput.
- A lot of small simple cores, running at low speeds.
- Lots of ALUs for computation, little hardware for control..

The best is to combine these two.

11.4 General Purpose GPUs

Very efficient for:

- Fast parallel floating point processing.
- SIMD (Single Instruction Multiple Data) operations.
- High computation per memory access.

Not as efficient for

- Double precision computations.
- Logical operations on integer data.
- Random access, memory-intensive operations.
- Branching-intensive operations

GPU control unit fecthes **1** instruction per clock cycles, and shares it with **8** processors.

11.4.1 Divergent Execution

This is what happens when all the processors mentioned above doesn't want the same instruction. This would really hurt the performance.

An example of this is given in figure 25

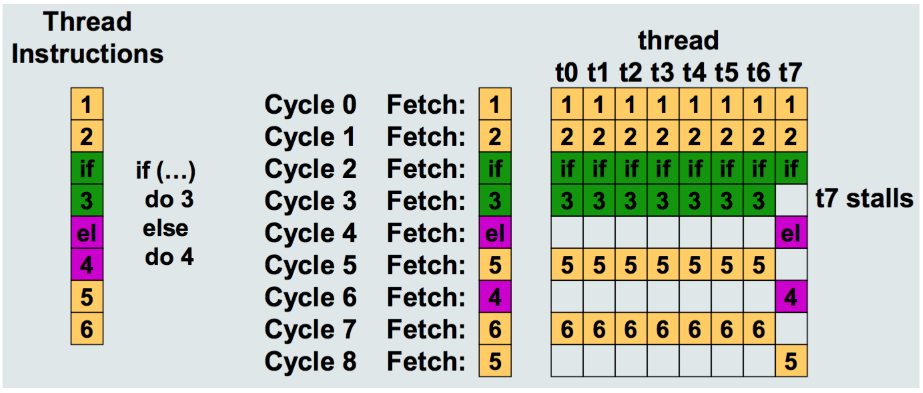


Figure 25 – Divergent execution due to a branch.

11.4.2 Reduce Branch Divergence

We can reduce these with some software optimizations, such as:

- target a divergent branch enclosed by a loop.
- execute loop iterations that take the same branch direction.
- delay those that take the other direction until later iterations.

- execute the delayed operations in future iterations where more threads take the same direction.

11.4.3 GPUPU

General-purpose computing on graphics processing units is the use of a GPU for doing computations usually handled by the CPU. We can't use any GPU though, it needs to have been provided hardware support to be able to do this. One such GPU is the Intel Tesla.

11.4.4 NVIDIA Tesla

Its massively parallel with 1000s of processors, aswell as cost and power efficient.

11.4.5 CUDA Programming Language

Computer Unified Device Architecture is a parallel computing platform and application programming interface (API) model created by NVIDIA. It allows software developers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing.

It uses an heterogeneous programming approach, serial computation will be handled as usual by the cpu, but parallel computations that are normally handled by the CPU can now be sent to the GPU. An example can be found in the computer game industry, where GPUs are used not only for graphics rendering but also in game physics calculations (physical effects such as debris, smoke, fire, fluids); examples include PhysX and Bullet.

CUDA can not only be used in GPU computing but also in multi-core CPUs, and works especially well with C, C++ and Fortran.

General CUDA steps:

1. Copy data from CPU to GPU.
2. Perform SIMD computations on GPU.
3. Copy back data from GPU to CPU.

We want to minimize data transfer between CPU & GPU and maximize the number of threads on the GPU, the execution on host doesn't wait for kernel to finish on GPU.

xb

12 Lecture 12: Low Power Architecture

For some applications, low power consumption is more important than performance:

- Mobile communication and computing
- Wirteless internet
- Medical implants
- Deep space applications

This will lead to longer battery life and to keep costs down, in terms of power consumptions. It will also lead to a longer life due to lower temperature, as can be seen in figure 26.

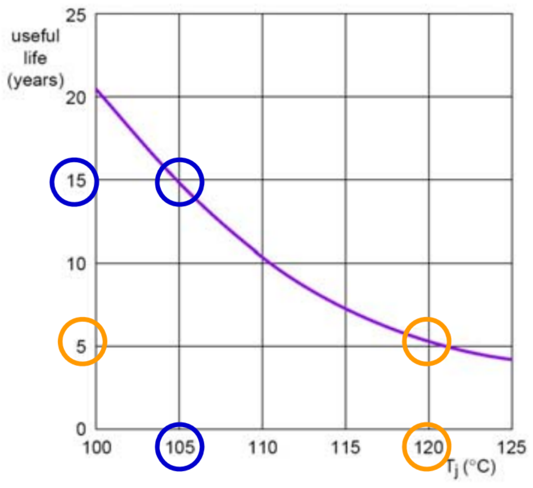


Figure 26 – How the temperature affects the lifespan of a processor..

12.1 Design for low power kanske kan plocka bort denna och låta allt stå under första rubriken

12.2 The Cursoe Processors

12.3 The ARM Processors

12.4 Final remarks