



Topic Notes: Modern Architecture Theme: Parallelism

The increases in processing power for decades have come from, at least in part, faster and faster clock speeds.

- we have seen in this course some of the reasons for limitations – gate delays
- smaller components means shorter gate delays, allowing shorter clock cycles and faster processors
- as we approach the physical limitations of the sequential processor, performance gains are coming more and more from the exploitation of parallelism
- there are many ways to expose native concurrency and introduce explicit parallelism to our processors, and we'll look at a few today

Instruction-Level Parallelism

We begin with the topics in P&H Section 4.10 – *instruction-level parallelism* (ILP).

The idea here is that we take a sequence of instructions that are intended to be executed one at a time and in sequence and attempt to overlap their execution. We need to take care to ensure that any parallelism we introduce will produce the same result as the one at a time, sequential execution of those instructions.

We have seen one very common method of exploiting parallelism with this approach – pipelining.

We saw how pipelining can improve the throughput of instructions for a processor, at the expense of some additional hardware. We also saw how much care is required to ensure correctness in this case, dealing with hazards through data forwarding and pipeline stalls.

One way to achieve this is to try to lengthen our pipelines, breaking down the slower stages into multiple stages to allow each stage to be shorter and to allow more instructions to be executing in parallel in the pipeline.

Another common approach to instruction-level parallelism involves launching more than one instruction at each pipeline stage – a technique called *multiple issue*.

.

The goal here is to be able to issue (start/complete) more than one instruction per cycle. The range in modern processors is 3-6 instructions issued per clock cycle.

Our text breaks down multiple issue into two main categories:

1. *static multiple issue* – where the parallelism is determined, at least in part, statically by a compiler
2. *dynamic multiple issue* – where parallelism is determined at run-time by the processor

In either case, instructions are organized into *issue slots*, which are “starting positions” in preparation for entry into the pipeline.

Instructions need to be organized so that can be executed in parallel with each other (and with other instructions already in the pipeline) without interference.

We saw how important and potentially complex data and control hazards can be even in our 5-stage single-issue pipeline – adding ILP provides more “opportunity” for hazards.

Dealing with the potential hazards is a collaborative effort between a compiler that would produce code that avoids hazards and hardware that detects them and deals with them appropriately (using the techniques we discussed earlier).

An important idea in achieving an effective ILP is that of *speculation*. Here, we make some guesses or assumptions about instructions to keep our multiple pipelines full in those cases where the guesses turn out to be true.

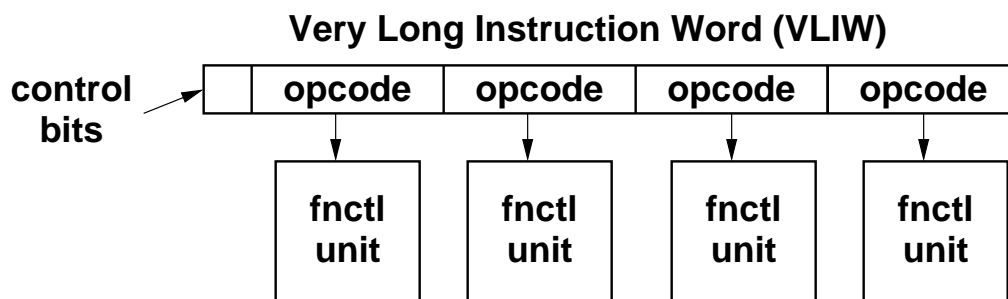
Speculation might include:

- conditional branch results
- that sequences of instructions do not have data dependencies

However, speculation can be incorrect so mechanisms must be included to deal with those cases where problems arise.

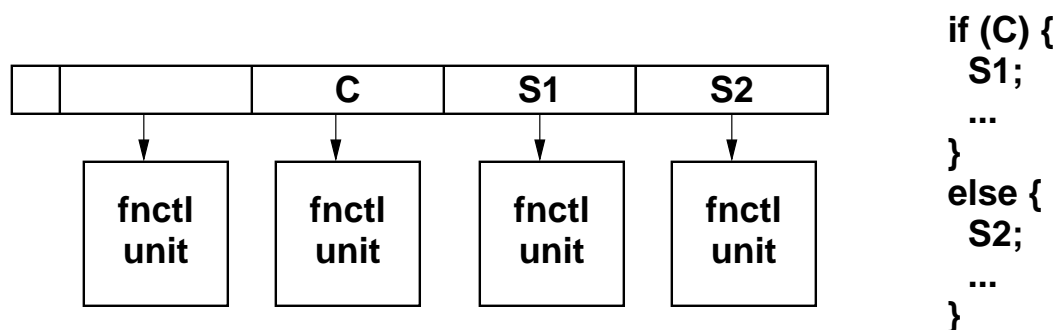
Static Issue/Explicitly Parallel Instruction Computers

Static multiple issues has also been referred to as the *explicitly parallel instruction computer (EPIC)* or *very long instruction word (VLIW)* machine. Examples include Yale’s ELI and Intel’s Itanium.



- New architectural approach

- An *issue packet* of instructions or *molecule* is made up of several concurrently executed *atoms*
- Each atom is assigned to a separate *functional unit*
- Processor may have additional registers, each atom gets its own copy of registers that are *committed* only when the atom is retired
- A few (8 of 128) bits of the molecule govern the execution of the atoms
- Speculative execution: avoid conditional branch overhead—execute then *and* else, but commit only one (Disadvantage: some work is guaranteed to be wasted)



- Very complex programming – meant to be done by compilers, not people

P&H Figures 4.68 and 4.69 show a simple two-issue MIPS processor.

- rule: each issue packet can contain at most one ALU/Branch instruction and one load/store instruction
- the compiler groups instructions when possible – insert nops when no appropriate instruction is available to group
- the compiler may be held responsible for removing all hazards to simplify the hardware
- even without additional forwarding and/or hazard detection, additional hardware is needed to permit more values to be read/written to the register file on each cycle and to be able to compute both ALU result or a branch target at the same time as an effective address for a memory access

Figure 4.70 shows how the following loop could be adapted to the two-issue setup:

```

Loop:  lw    $t0, 0($s1)      # temp = array elt
      addu $t0, $t0, $s2   # add $s2 val to temp
      sw   $t0, 0($s1)    # array elt = temp
      addi $s1, $s1, -4   # advance to prev array elt
      bne  $s1, $zero, Loop # branch if s1!=0

```

In Figure 4.70, we see a very disappointing situation - there is little opportunity to make use of our two-issue system.

Figure 4.71 shows a much more effective translation of this loop to the two-issue system. Here, a technique called *loop unrolling* is used to generate multiple copies of the code in the loop, so each iteration of the generated code performs multiple passes of the original loop. This gives much more opportunity for rescheduling the instructions to pair up ALU/branch instructions with memory ops.

Dynamic Multiple Issue/Superscalar Processors

Moving away from the compiler and toward the hardware, we can consider a *superscalar* processor that is capable of starting multiple “regular” instructions (not a compiler-generated issue packet) on each clock cycle.

In its simplest form, the hardware will consider the next few instructions (in order) and quickly determine how many may safely be executed in parallel (avoiding hazards), and will issue that collection of instructions as a group.

Figure 4.72 shows an example of *dynamic pipeline scheduling* where a three-stage pipeline is used to collect groups of instructions to be executed (in *reservation stations*), execute them in parallel and/or out of order, using multiple functional units (maybe 10-12 at a time), and a *commit unit* that writes back the results of the instructions in order.

Figure 4.73 shows some statistics about the parallelism in older and recent Intel and Sun processors.

Intel Pentium Parallel Extensions

You may have heard of the MMX (and AMD’s 3DNow!, and more recently SSE, SSE2, SSE3, SSE4, and SSE5) extensions to the Intel Pentium core.

- SSE = Streaming SIMD Extensions, SIMD = single instruction multiple data
- Very simple idea to support arithmetic on short operands: cut the carry lines – makes it possible to manipulate 8 bytes (64 bits) independently but simultaneously in a single instruction
- packing and unpacking instructions are needed
- relatively few changes to the ALU, and we get the manipulation of two RGB+ α pixels in a single operation

```
rrrrrrrrr gggggggg bbbbbbbb alphalph rrrrrrrrr gggggggg bbbbbbbb alphalph
```

treat as 8, 8-bit numbers

- several modes provided allow the ALU to consider its input as 8 independent bytes or 4 independent 2-byte values or 2 independent 4-byte values

- makes programming more challenging: consider conditional pixel modification

Multicore Architectures

The recent approach involves replicating processing “cores” on the same chip that traditionally held a single processor.

This is the *multicore* or *symmetric multithreaded (SMT)* approach.

This has changed the nature of the increases in processing capabilities:

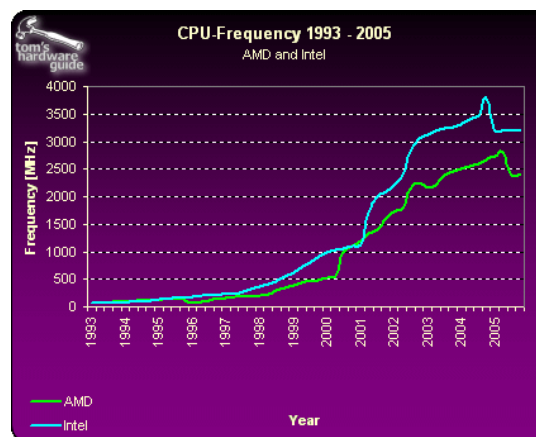


Figure used with permission from article *The Mother of All CPU Charts 2005/2006*, Bert Töpelt, Daniel Schuhmann, Frank Völkel, Tom’s Hardware Guide, Nov. 2005,
http://www.tomshardware.com/2005/11/21/the_mother_of_all_cpu_charts_2005/

We will look briefly at the architecture – programming these is a nightmare for another day.

Intel/AMD Multicore

Intel and AMD have both introduced a series of chips that contain multiple processing cores.

The Intel Core Duo:

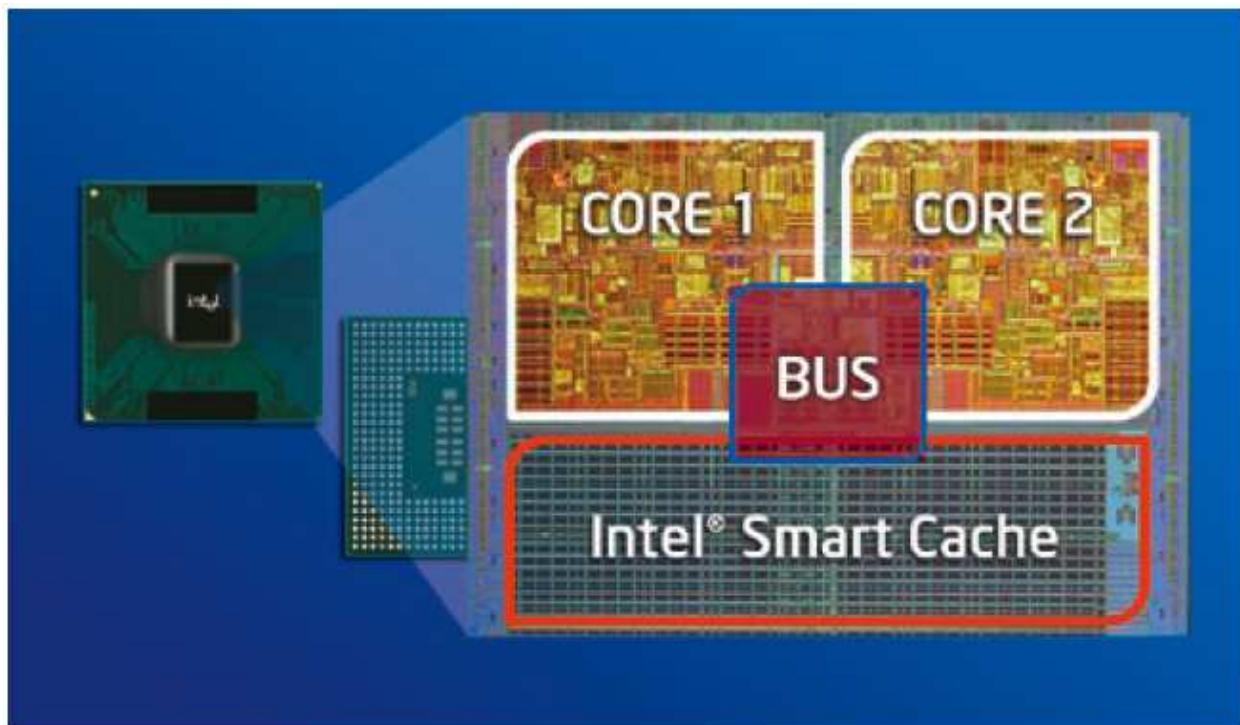


Image from Intel Core Duo Processor product brief

- Independent copies of ALU, registers, L1 cache
- Processors on the same chip share L2 cache
- This will require some cache coherency protocol
- Up to the operating system to schedule processes/threads to keep each core occupied

Cell Broadband Engine

IBM, Sony, and Toshiba collaborated on the recent *Cell* architecture.

- Consists of one or more PowerPC Processor Elements (PPEs) that are like traditional processors, and several Synergistic Processor Elements (SPEs) that are simpler processors that only perform work as assigned to them by PPEs
- Instructions and the data they manipulate are bound together in an *apulet*
- A *cell* is a hierarchically structured “bundle of control and streaming processor resources” or scalable *processing element*
- Apulets can be arbitrarily assigned to cells
- More intense computation is performed by adding more cells to the pool

- Currently used in Playstation 3
-

Graphics Processing Units

Computer graphics has driven the development of modern SIMD (single instruction multiple data) processors used as Graphics Processing Units.

- Graphics computations are often applied to a group of pixels at the same time – hence the SIMD approach – you can process many pixels at once (typically 128), but you have to do exactly the same operation on each
 - Typically restricted to the single-precision floating-point operations needed for graphics
 - Focus on maximizing “frames per second”
 - Operations use graphics terminology: “pixel shaders” or “vertex shaders”
 - But...deliver hundreds of gigaflops of performance where traditional CPUs are in the tens at best
 - People have noticed this performance and have harnessed this computational power for non-graphics applications
 - GPU producers have noticed this interest and are now providing better programming capabilities and double-precision operations (needed for most serious scientific calculations)
-

Cache Coherency

Any time a set of processors share a common memory but have private caches at some point in the memory hierarchy, the issue of *cache coherency* will arise. Once an issue only of concern to high-end multiprocessors, it is now the concern of every computer with a multi-core processor chip.

Unlike the case of a single-processor memory hierarchy, a multiprocessor with private caches at a low level will allow a block of memory may reside in a line in more than one processor’s cache. This is not a problem if neither processor modifies the memory. But consider this situation:

We have two processors, each with its own private L1 cache. The programs running on the CPUs both access the same memory location, x . The following sequence of actions occurs:

1. x is initialized to 0, and after some period of time, neither cache contains the block that includes x
2. CPU 0 reads x , and x ’s block is brought into a line of CPU 0’s L1 cache

3. CPU 1 reads x , and x 's block is brought into a line of CPU 1's L1 cache
4. CPU 0 writes $x=1$ into the location in its cache
5. CPU 1 reads x

What value of x will be seen by CPU 1? It had better be 1, but it might not be unless there is a mechanism to ensure that it is.

We want to maintain *coherency* and *consistency* with our caches.

A memory hierarchy is coherent if

1. A read of a memory location by processor P following a write of that memory location by processor P (with no intermediate writes) will return the value written by P
2. A read of a memory location by a processor P following a write of that memory location by another processor P' returns the value written by P' if some minimum amount of time has passed between the write by P' and the read by P
3. Writes are *serialized* – the values written over time by any processors are seen in that same order by all processors

The first two are pretty straightforward. The last ensures that all processors see the most recently written value by any processor.

How can we make this happen, while maintaining efficient cache operation (which is essential to efficient processing)?

A *cache coherence protocol* must be implemented to ensure correctness.

A popular protocol is based on *snooping*. All caches watch a shared memory access bus to determine whether local cache lines are also in the caches of other CPUs.

A snooping protocol can be used to implement a *write invalidate protocol*. Here, any time a CPU writes to memory, it makes sure that it has the only copy of the cache line that contains the memory to be modified. It does this by writing an “invalidate” message on the shared bus, which will cause all other caches to mark that cache line as invalid (if they have a copy). Thus, if and when another CPU attempts to access that cache line, it will get a cache miss and will fetch the (updated) block from memory.

A cache coherence protocol that is correct, fast, and scalable is a key component of modern multi-processor design, and would certainly be an important topic in an advanced architecture course.