

The EPIC Architecture

Colin Tan

S15-04-26

ctank@comp.nus.edu.sg

EPIC – An Introduction

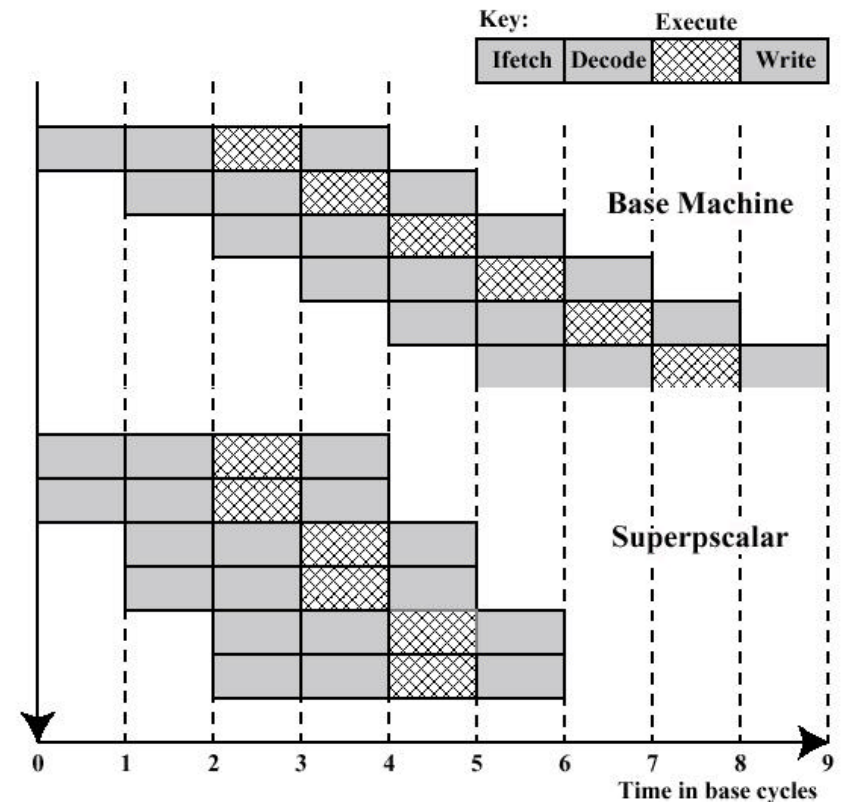
- EPIC
 - Explicitly Parallel Instruction Computing
 - Instruction Level Parallelism (ILP) is identified to hardware by the compiler.
 - Particular EPIC architecture we cover today: IA64
- Topics Covered Today
 - Motivations behind IA64
 - Difficulties in exposing ILP in practical code
 - Data Hazards
 - Control Hazards
 - Memory Latency
 - Practical IA64 Implementation
 - The Itanium Processor

A Little Background

- Improved integrated circuit fabrication over the last 20 years
 - Smaller feature sizes, multi-layer fabrication.
 - Items closer together, reducing signal propagation. Faster circuits.
 - Able to squeeze more onto the same die real estate!
- Natural inclination:
 - Take advantage of extra potential real-estate to introduce multiple copies of (pipelined) functional units.
 - Integer units, branch units.
 - Multiple functional units => multiple execution streams
 - We can now do more than one instruction per clock cycle than before!
- Most common implementation of this: The Superscalar Architecture.
- Other less popular architectures like VLIW omitted today.

Background: Superscalar Processing

- Base Machine
 - Single pipeline
 - Execute at most one instruction per clock cycle after steady state.
- Superscalar Machine, Two Functional Units
 - Execute two instructions per clock cycle after steady state.
 - 100% improvement over Base Machine.

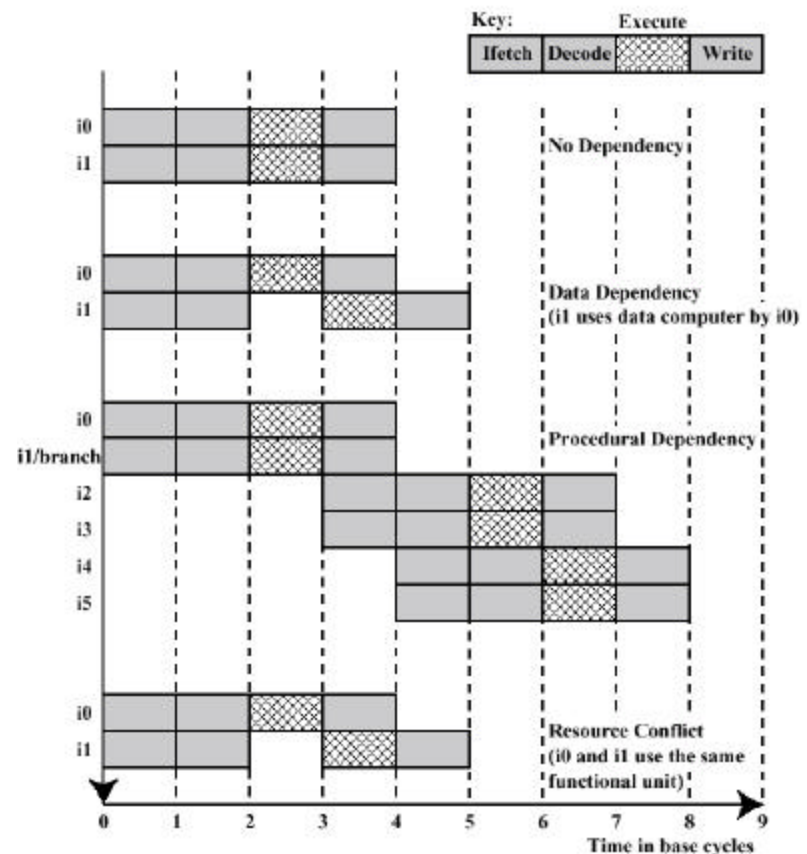


Limitations to Superscalar ILP

- The ideal picture shown earlier is marred by several problems:
 - Data Hazards
 - Data dependencies between instructions mean that these instructions cannot be executed correctly together.
 - Control Hazards
 - ILP identification requires several instructions to be pre-fetched to find parallel instruction pairs, as well as to keep pipelines full.
 - Branches make this difficult
 - What if we pre-fetch from the wrong side of a branch?
 - Memory Latencies
 - Load-Use dependencies cause pipelines to stall.
 - Stalled pipelines cannot execute further instructions till stall is resolved.
 - Can take between one and several hundred cycles to resolve!

Effect of Hazards on Superscalar Execution

- Figure shows effects of hazards
 - End result in each case is that fewer than two instructions are completed per clock cycle.
 - Loss of efficiency
 - This means loss of processing power!



Data Hazards

- Data hazards limit ILP. E.g.
 - I1: movi r3, 3*
 - I2: add r1, r3, r4*
 - Instructions I1 and I2 cannot possibly execute together correctly even if multiple integer units are available.
- Superscalar Architectures must provide logic to determine if such dependencies exist.
 - Pre-fetch a small number (e.g. 16) of instructions.
 - Have circuits to determine dependencies between instructions.
 - Issue instructions that are independent.
- This is a complex, expensive process.
 - Need skilled engineering designs => Expensive
 - Occupies precious chip real-estate => Expensive
 - Complex circuits => Higher Failure Rate => Expensive

Data Hazards

The IA64 Solution

- Idea: Pass the responsibility to the compiler
 - Easier to find ILP using software algorithms than digital hardware circuits.
 - Does not consume chip real-estate.
 - Compiler can search for independent instructions across entire sub-program.
 - Hardware-based solutions can only find within small pre-fetch window.
- Much greater chance of finding ILP at much lower cost.

Data Hazards

The IA64 Solution

- Compiler will identify instructions that can be executed in parallel and inform the hardware.
 - IA64 defines 128-bit bundles
 - 3 instructions (“packs”) of 40 bits each
 - 8 bit Template
 - Dependent and independent instructions may be mixed in the same bundle
 - Compiler sets template bits to inform hardware which instructions are independent.
 - Template can identify independent instructions across bundles.
 - Instructions in bundles do not have to be in original program textual order.
- This system maximum utilization of available functional units.

Control Hazards

- Consider this program:

I1: mov r2, r3

I2: sub r1, r2, r7

I3: beq r1, I7

I4: sub r3, r1, r7

I5: muli r3, r3, 2

I6: j I10

I7: muli r7, r3, 4

I8: div r6, r7, r3

I9: shri r6, 7

I10:

- When we encounter the branch in I3:
 - Should we start pre-fetching and executing I4 ... I6? **OR**
 - Should we start pre-fetching and executing I7 ... I9?
- This decision is made before the branch outcome is known, to prevent stalls. Mistakes are costly.
 - Must unroll instructions erroneously executed.
 - Erroneous execution wastes time.

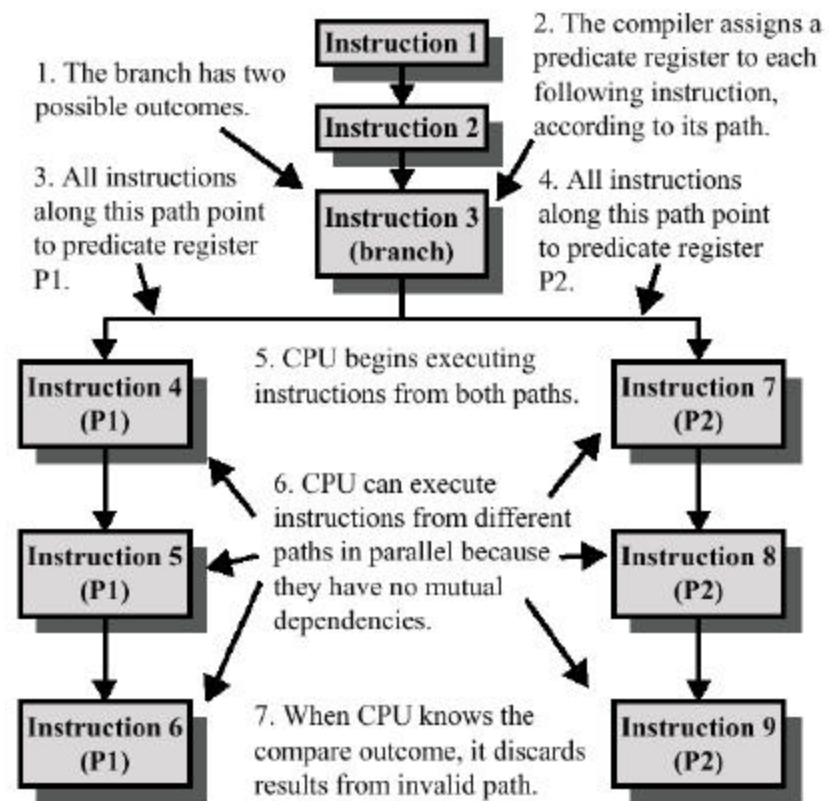
Control Hazards

The IA64 Solution

- IA64 solves this by predication.
- Idea:
 - Compiler tags each side of a branch with a predicate.
 - Bundle tagged instructions and set template bits to allow parallel execution of predicated instructions.
 - Both sides of the branch are executed simultaneously.
 - When the outcome of the branch is known, the effects of the correct side are committed (registers modified, etc.), while the effects (and remainder) of the wrong side are discarded.
- Benefits:
 - No need to unroll effects (they are committed only after we know which is the correct side the branch)
 - Time taken to execute wrong side at least partially amortized by execution of correct side, assuming sufficient functional units.

Predication Example

I1: mov r2, r3
I2: sub r1, r2, r7
I3: P1, P2 = cmp(r1 == 0)
I4: <P1> sub r3, r1, r7
I5: <P1> muli r3, r3, 2
I6: <P1> j I10
I7: <P2> muli r7, r3, 4
I8: <P2> div r6, r7, r3
I9: <P2> shri r6, 7
I10:



Memory Latencies

- Memory Units are Slow
 - Delay of between one and several tens of thousands of cycles, depending on multi-level cache hit/miss profiles.
- E.g.
 - I1: lw r1, 0(r4)
 - I2: add r2, r1, r3
- Potentially delay of several thousand cycles between I1 and I2. May be impractical to schedule sufficient instructions in between.

Memory Latencies

The IA64 Solution

- Idea: Move all load instructions to the start of the program.
 - This is called “hoisting”.
 - Loads will be executed concurrently with the rest of the program.
 - Hopefully data will be ready in register when it is read.
 - Loads may belong to decision paths that are never executed.
 - Hoisting effectively causes these loads to be executed anyway, even if their contents aren’t actually required.
 - Loads are therefore “speculative”: They are done on the speculation that the results are needed later.
 - A “Check” instruction is placed just before the load results are needed.
 - This checks for exceptions in the speculative load.
 - As well as commits effect of load to the target register.

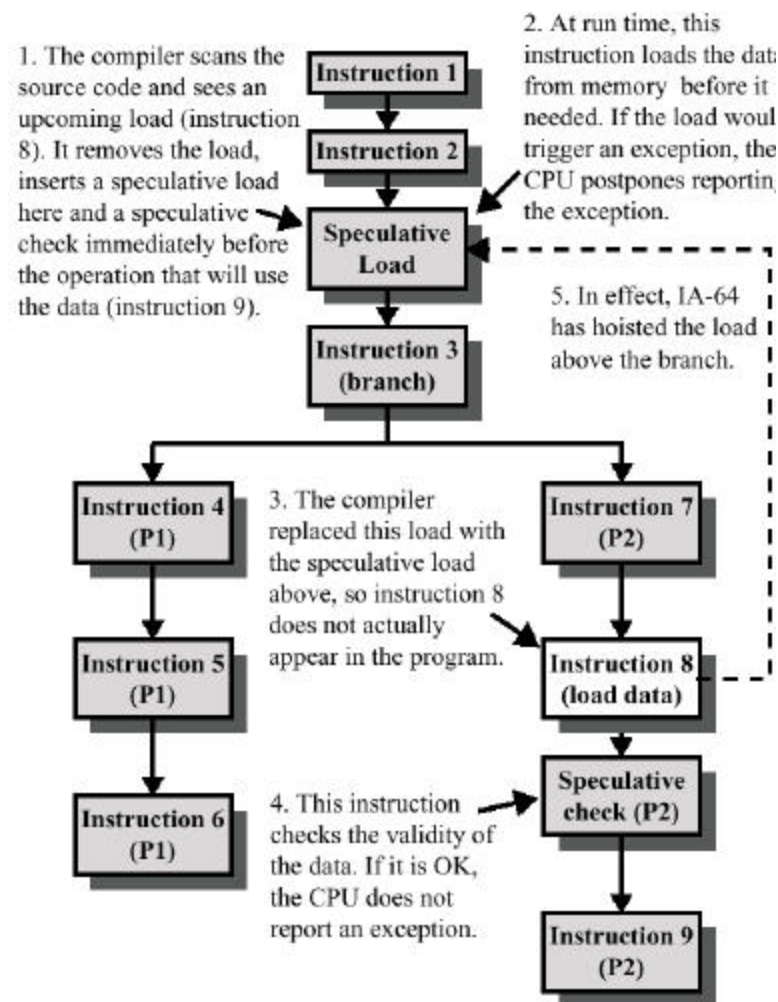
Speculative Load Example

- Original code:

I1: add r3, r1, r2
I2: sub r2, r3, r5
I3: P1, P2 = cmp(r2 == 3)
I4: <P1> sub r7, r3, r4
I5: <P1> rori r7, 3
I6: <P1> j I10
I7: <P2> lw r4, 0(r6)
I8: <P2> muli r4, r3, r6

- Modified Code:

I1: add r3, r1, r2
I2: sub r2, r3, r5
I3: lw r4, 0(r6) ← Speculative Load
I4: P1, P2 = cmp(r2 == 3)
I5: <P1> sub r7, r3, r4
I6: <P1> rori r7, 3
I7: <P1> j I10
I9: <P2> chk.s r4 ← Load Check
I10: <P2> muli r4, r3, r6 ← Load used here



Other Nice IA64 Features

- Transferring ILP discovery to compilers frees up chip real-estate.
- Real estate rededicated to:
 - Huge GP and FP register files of 128 registers each.
 - Larger caches, with more read/write ports, greatly reducing memory latency.
 - More functional units (hence making predicated execution feasible).

IA-64 Implementation

The Itanium Processor

- Itanium (Merced) Features:
 - System Bus:
 - 64 bits wide
 - Core 800 MHz
 - 2.1 GB/s transfer
 - Width
 - 2 bundles per cycle
 - 4 integer units
 - 2 load *or* stores per clock
 - 9 instruction issue ports.
 - Caches
 - L1 – 2 x 16KB, 2 clock latency
 - L2 – 96K, 12 clock latency
 - L3 – 4MB external, 20 clock latency, 11.7 GB/s bandwidth

IA-128 Implementation

The Itanium 2 Processor

- Itanium 2 (McKinley) Features:
 - System Bus:
 - 128 bits wide
 - Core 1.0 GHz
 - 6.4 GB/s transfer
 - Width
 - 2 bundles per cycle
 - 6 integer units
 - 2 load *and* stores per clock
 - 11 instruction issue ports.
 - Caches
 - L1 – 2 x 16KB, 1 clock latency
 - L2 – 256K, 5 clock latency
 - L3 – 3MB on-die, 12 clock latency, 32 GB/s bandwidth

Future Implementation

The Itanium 2-2003 Processor

- Itanium 2-2003 (Madison) Features:
 - System Bus:
 - 128 bits wide
 - Core 1.5 GHz
 - 6.4 GB/s transfer
 - Width
 - 2 bundles per cycle
 - 6 integer units
 - 2 load *and* stores per clock
 - 11 instruction issue ports.
 - Caches
 - L1 – 2 x 16KB, 1 clock latency
 - L2 – 256K, 5 clock latency
 - L3 – 6MB on-die, 14 clock latency, 48 GB/s bandwidth, doubled set associativity.

Summary

- Limitations of Superscalar Processor
 - Complicated hardware logic to detect dependencies between a tiny buffer of instructions.
 - Branches require prediction to facilitate pre-fetching. Wrong predictions are expensive.
 - Memory latencies further exacerbate situation.
- IA64 Solutions:
 - Move complication of finding ILP to compiler.
 - Easier to do in software.
 - Freed-up real-estate used to boost resources of processor.
 - Predication (coupled with increased resources), eliminate costs of branch prediction.
 - Memory latencies ameliorated by speculated loading.

References

- *Computer Organization and Architecture, 5th Edition*, William Stallings, Prentice Hall International Editions, ISBN 0-13-085263-5, 2000.
- *Overview of the IPF Architecture*, Hewlett Packard Company, 2002.
- *Next Generation Itanium Processor Overview*, Gary Hammond, Sam Naffziger, Intel Developer Forum, Fall 2001.
- *A 1.5GHz Third Generation Itanium Processor*, Jason Stinson, Stefan Rusu, Intel Corporation, 2002.