Chapter Four – II (5/5)

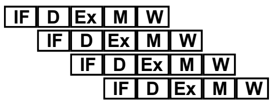
Instruction-Level Parallelism (ILP)

- **Pipelining: executing multiple instructions in parallel**
- **To increase ILP**
  - **Deeper pipeline**
    - Less work per stage ⇒ shorter clock cycle
  - **Multiple issue**
    - Replicate pipeline stages ⇒ multiple pipelines
    - Start multiple instructions per clock cycle
    - CPI < 1, so use Instructions Per Cycle (IPC)
    - E.g., 4GHz 4-way multiple-issue
      - 16 BIPS, peak CPI = 0.25, peak IPC = 4
    - But dependencies reduce this in practice

Issues in Pipelined design

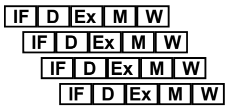
Limitation

- **Pipelining**



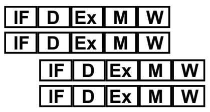
Issue rate, FU stalls, FU depth

- **Super-pipeline**
  - **- Issue one instruction per (fast) cycle**
  - **- ALU takes multiple cycles**



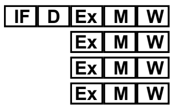
Clock skew, FU stalls, FU depth

- **Super-scalar**
  - **- Issue multiple scalar instructions per cycle**



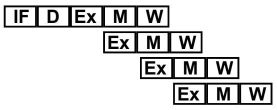
Hazard resolution

- **VLIW ("EPIC")**
  - **- Each instruction specifies multiple scalar operations**
  - **- Compiler determines parallelism**



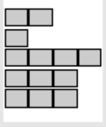
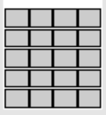
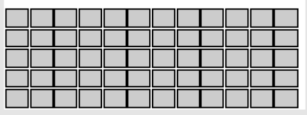
Packing

- **Vector operations**
  - **- Each instruction specifies series of identical operations**

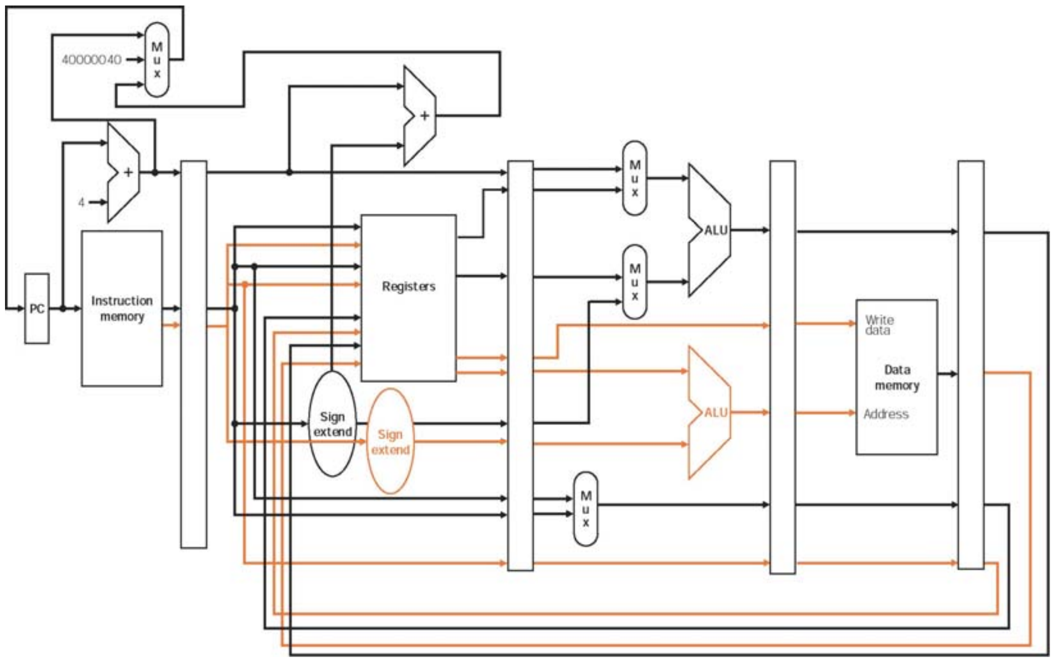


Applicability

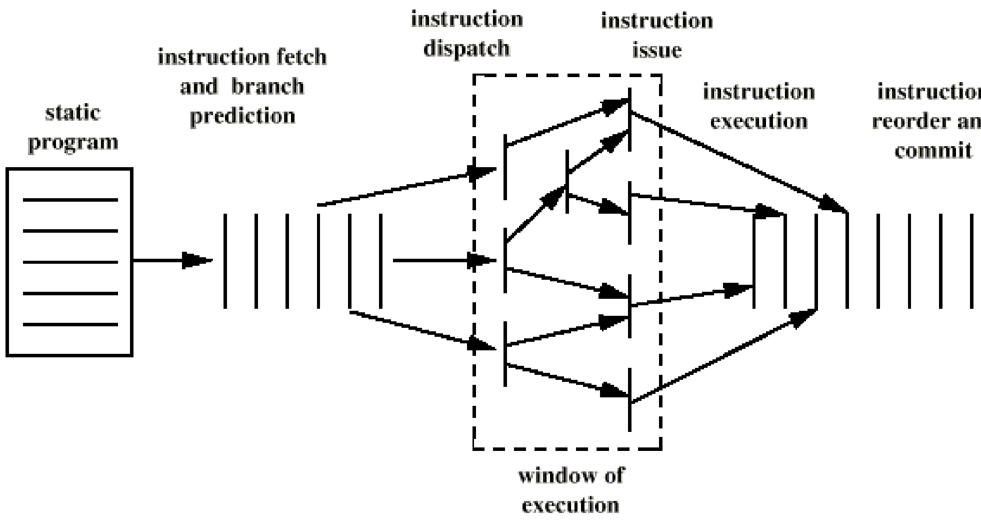
Comparison: CISC, RISC, VLIW

ARCHITECTURE CHARACTERISTIC	CISC	RISC	VLIW
INSTRUCTION SIZE	Varies	One size, usually 32 bits	One size
INSTRUCTION FO RMAT	Field placement varies	Regular, consistent placement of fields	Regular, consistent placement of fields
INSTRUCTION SEMANTICS	Varies from simple to complex; possibly many dependent operations per instruction	Almost always one simple operation	Many simple, independent operations
REGISTERS	Few, sometimes special	Many, general-purpose	Many, general-purpose
MEMORY REFE RENCES	Bundled with operations in many different types of instructions	Not bundled with operations, i.e., load/store architecture	Not bundled with operations, i.e., load/store architecture
HARDWARE DESIGN FOCUS	Exploit microcoded implementations	Exploit implementations with one pipeline and & no microcode	Exploit implementations with multiple pipelines, no microcode & no complex dispatch logic
PICTURE OF FIVE TYPICAL INSTRU CTIONS ▭ = 1 BYTE			

Superscalar pipelining



Superscalar Execution



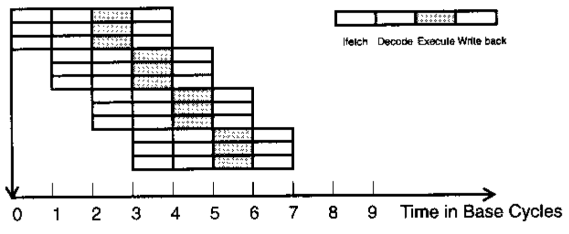
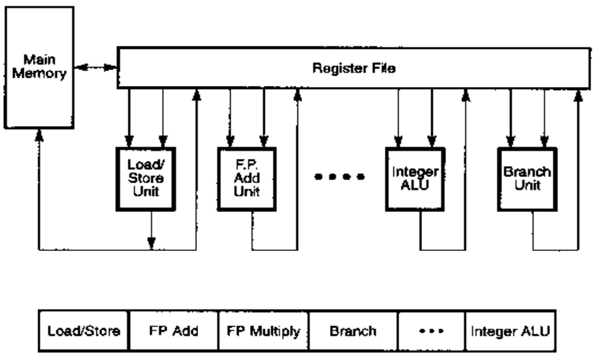
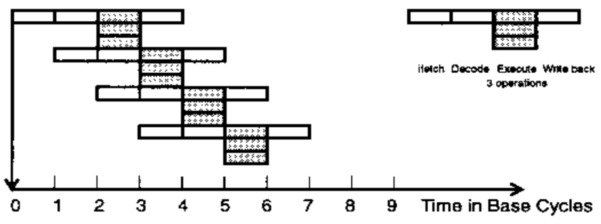


Figure 4.11 A superscalar processor of degree m = 3.

VLIW Execution



(a) A typical VLIW processor and instruction format



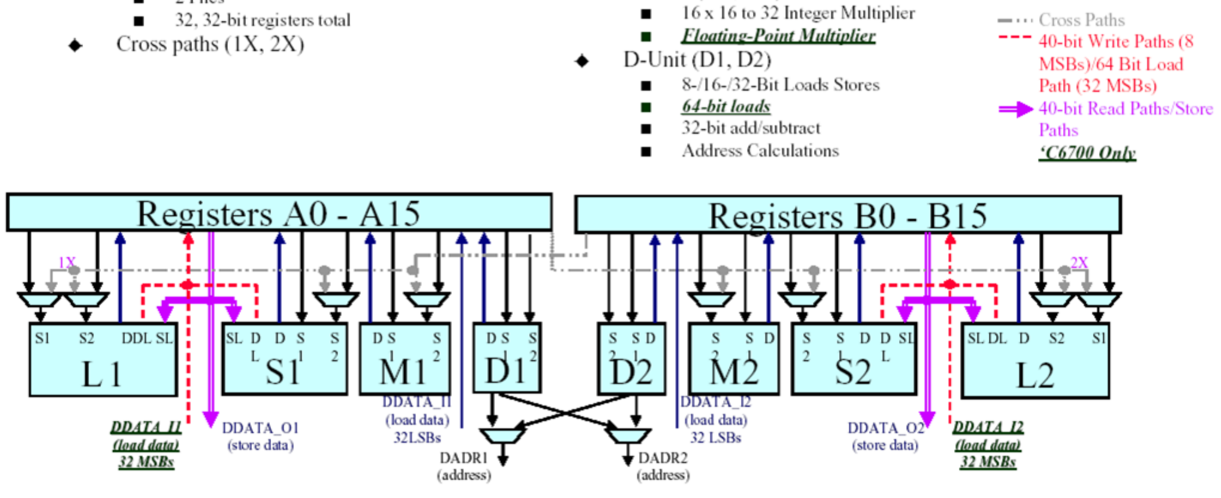
(b) VLIW execution with degree $m = 3$

Figure 4.14 The architecture of a very long instruction word (VLIW) processor and its pipeline operations. (Courtesy of Multiflow Computer, Inc., 1987)

TMS320C6000 CPUs

- 2 Data Paths
- 8 Functional Units
  - Upto 6 Integer ALUs
  - 2 Integer/***Floating Point*** Multipliers
  - ***2 Floating Point Arithmetic Units***
  - ***2 Floating Point Auxiliary Units***
- Control
  - Independent
  - Up to 8 32-bit Instructions
- Registers
  - 2 Files
  - 32, 32-bit registers total
- Cross paths (1X, 2X)

- L-Unit (L1, L2)
  - ***Floating-Point ALU***
  - 40-bit Integer ALU/Comparisons
  - Bit Counting
  - Normalization
- S-Unit (S1, S2)
  - ***Floating Point Auxiliary Unit***
  - 32-bit ALU/40-bit shifter
  - Bitfield Operations
  - Branching
- M-Unit (M1, M2)
  - 16 x 16 to 32 Integer Multiplier
  - ***Floating-Point Multiplier***
- D-Unit (D1, D2)
  - 8-/16-/32-Bit Loads Stores
  - ***64-bit loads***
  - 32-bit add/subtract
  - Address Calculations


Cross Paths
40-bit Write Paths (8 MSBs)/64 Bit Load Path (32 MSBs)
40-bit Read Paths/Store Paths
'C6700 Only
Registers A0 - A15
Registers B0 - B15
1X
2X
L1
S1
M1
D1
D2
M2
S2
L2
DDATA_I1 (load data) 32 MSBs
DDATA_O1 (store data)
DDATA_I1 (load data) 32LSBs
DADR1 (address)
DADR2 (address)
DDATA_I2 (load data) 32 LSBs
DDATA_O2 (store data)
DDATA_I2 (load data) 32 MSBs

Pros and Cons of VLIW

(+) Compiler prepares fixed packets of multiple operations that give the full "plan of execution"
- dependencies are determined by compiler and used to schedule according to function unit latencies
- function units are assigned by compiler and correspond to the position within the instruction packet ("slotting")
- compiler produces fully-scheduled, hazard-free code => hardware doesn't have to "rediscover" dependencies or schedule

(–) Compatibility across implementations is a major problem
- VLIW code won't run properly with different number of function units or different latencies
- unscheduled events (e.g., cache miss) stall entire processor

(–) Code density is another problem
- low slot utilization (mostly nops)
- reduce nops by compression ("flexible VLIW", "variable-length VLIW")

Dependencies Review

- Each of the three data dependencies
  - True data dependencies (read before write)
  - Antidependencies (write before read)
  - Output dependencies (write before write)

  $\left.\begin{array}{l}\text{Antidependencies}\\ \text{Output dependencies}\end{array}\right\}$ storage conflicts

  manifests itself through the use of registers (or other storage locations)
- True dependencies represent the flow of data and information through a program
- Anti- and output dependencies arise because the limited number of registers mean that programmers reuse registers for different computations
- When instructions are issued out-of-order, the correspondence between registers and values breaks down and the values conflict for registers

Antidependencies

- With OOE also have to deal with data antidependencies – when a later instruction (that completes earlier) produces a data value that destroys a data value used as a source in an earlier instruction (that issues later)

```
R3 := R3 * R5          True data dependency
R4 := R3 + 1           Output dependency
R3 := R5 + 1           Antidependency
```

- The constraint is similar to that of true data dependencies, except reversed
  – Instead of the later instruction using a value (not yet) produced by an earlier instruction (read before write), the later instruction produces a value that destroys a value that the earlier instruction (has not yet) used (write before read)

Storage Conflicts and Register Renaming

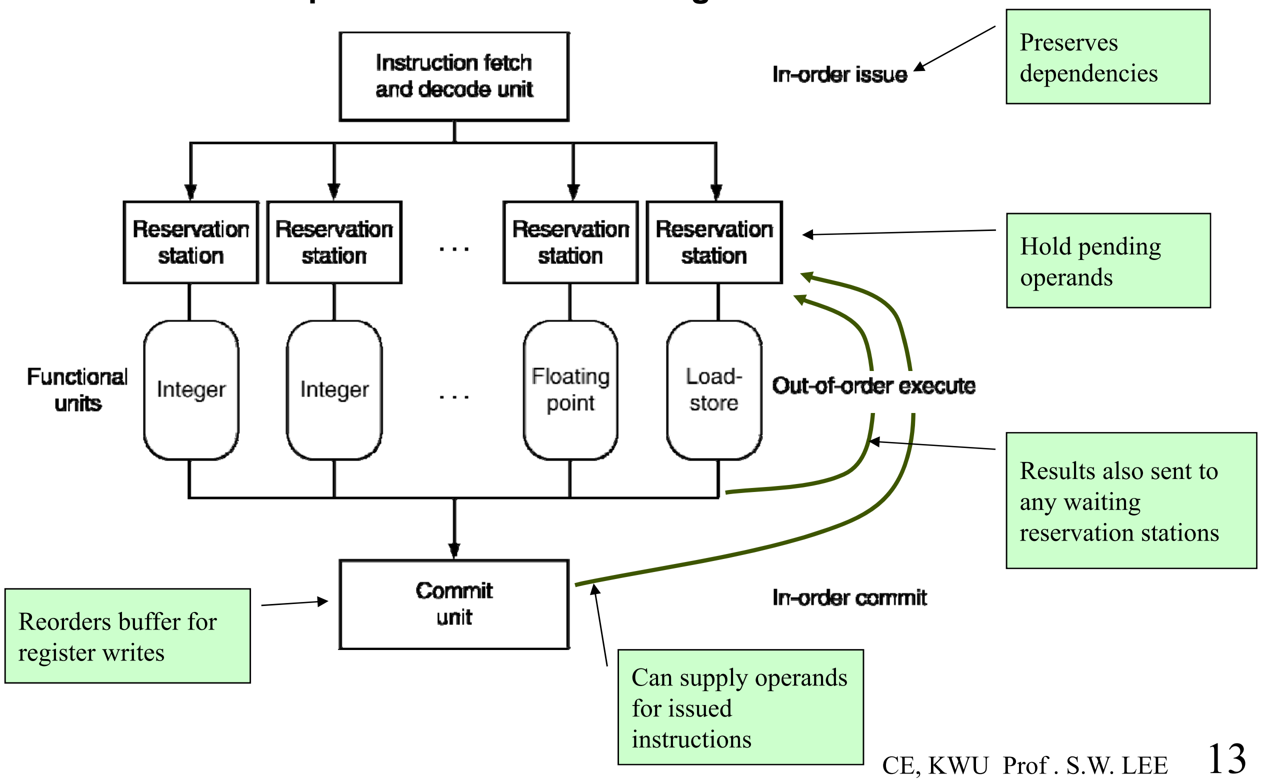
- Storage conflicts can be reduced (or eliminated) by increasing or duplicating the troublesome resource
  – Provide additional registers that are used to reestablish the correspondence between registers and values
    - Allocated dynamically by the hardware in SS processors
- Register renaming – the processor renames the original register identifier in the instruction to a new register (one not in the visible register set)

```
R3 := R3 * R5              R3b := R3a * R5a
R4 := R3 + 1      ⇨        R4a := R3b + 1
R3 := R5 + 1               R3c := R5a + 1
```

- The hardware that does renaming assigns a “replacement” register from a pool of free registers and releases it back to the pool when its value is superseded and there are no outstanding references to it

Dynamically Scheduled CPU: Superscalar

- The hardware performs the “scheduling”



Register Renaming

- **Reservation stations and reorder buffer effectively provide register renaming**
- **On instruction issue to reservation station**
  - **If operand is available in register file or reorder buffer**
    - Copied to reservation station
    - No longer required in the register; can be overwritten
  - **If operand is not yet available**
    - It will be provided to the reservation station by a function unit
    - Register update may not be required

Control Dependencies

- Every instruction is control dependent on some set of branches

```
if p1
    S1;
if p2
    S2;
```

- S1 is control dependent on p1, and S2 is control dependent on p2 but not on p1.
- control dependencies must be preserved to preserve program order

Branch Prediction and Speculative Execution

- Speculation is to run instructions on prediction – predictions could be wrong.
- Branch prediction: crucial to performance, could be very accurate
- Mis-prediction is less frequent event – but can we simply ignore?

Example:

```
for (i=0; i<1000; i++)
        C[i] = A[i]+B[i];
```

- Branch prediction: predict the execution as accurate as possible (frequent cases)
- Speculative execution recovery: if prediction is wrong, roll the execution back

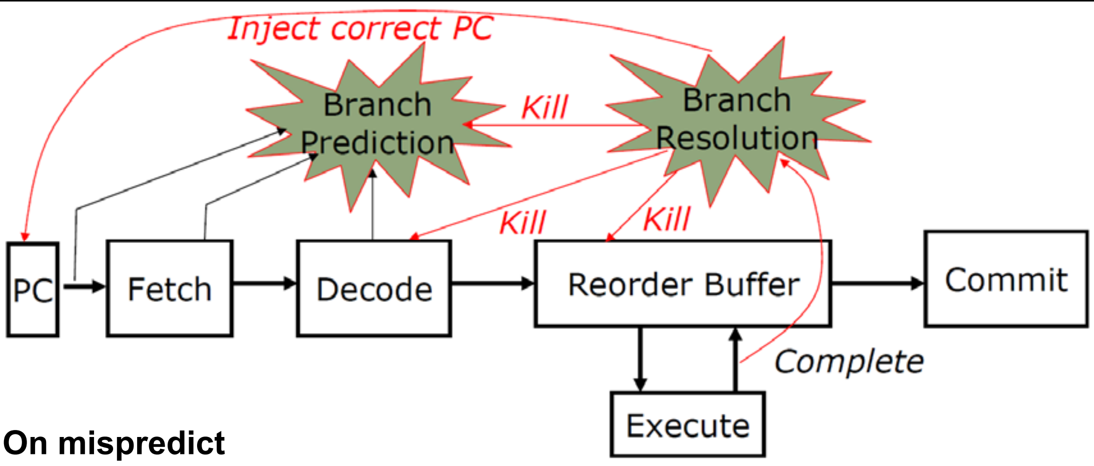
Speculation

- **Predict branch and continue issuing**
  - **Don't commit until branch outcome determined**
- **Load speculation**
  - **Avoid load and cache miss delay**
    - Predict the effective address
    - Predict loaded value
    - Load before completing outstanding stores
    - Bypass stored values to load unit
  - **Don't commit load until speculation cleared**

Mispredict Recovery

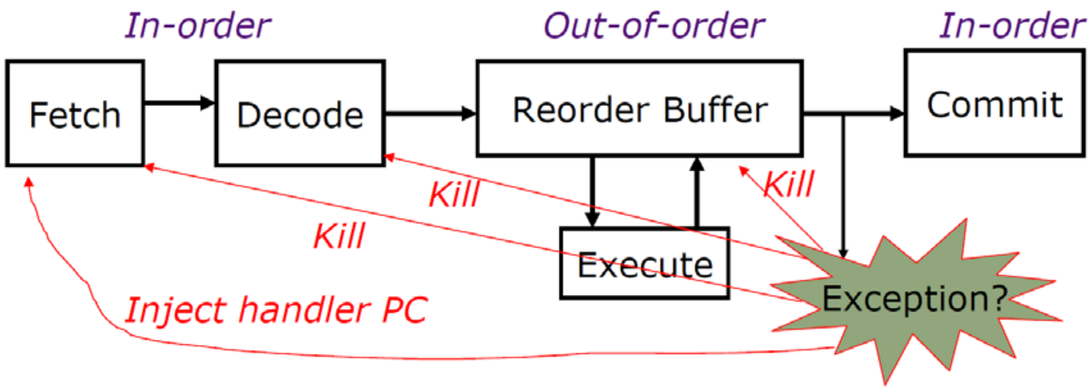
- **In-order execution machines:**
  - **Assume no instruction issued after branch can write-back before branch resolves**
  - **Kill all instructions in pipeline behind mispredicted branch**
- **Out-of-order execution?**
  - **Multiple instructions following branch in program order can complete before branch resolves**
  - **Out-of-order completion makes the results of mispredicted branches written on the register file.**
- **Now how can we roll back?**
  - **Do not write result to the register file before the branch is resolved**
  - **Use temporary storage (Reorder Buffer) for the results of speculative execution**

Branch Misprediction Recovery



- On mispredict
  - Roll back “next available” pointer to just after branch
  - Reset use bits
  - Flush mis-speculated instructions from pipelines
  - Restart fetch on correct branch path
- Can have multiple unresolved branches in ROB
- Can resolve branches out-of-order by killing all the instructions in ROB that follow a mispredicted branch

In-Order Commit for Precise Exceptions



- Instructions fetched and decoded into instruction reorder buffer in-order
- Execution is out-of-order (⇒ out-of-order completion)
- Commit (write-back to architectural state, i.e., regfile & memory, is in-order

➔ Temporary storage needed in ROB to hold results before commit

Speculating Both Directions

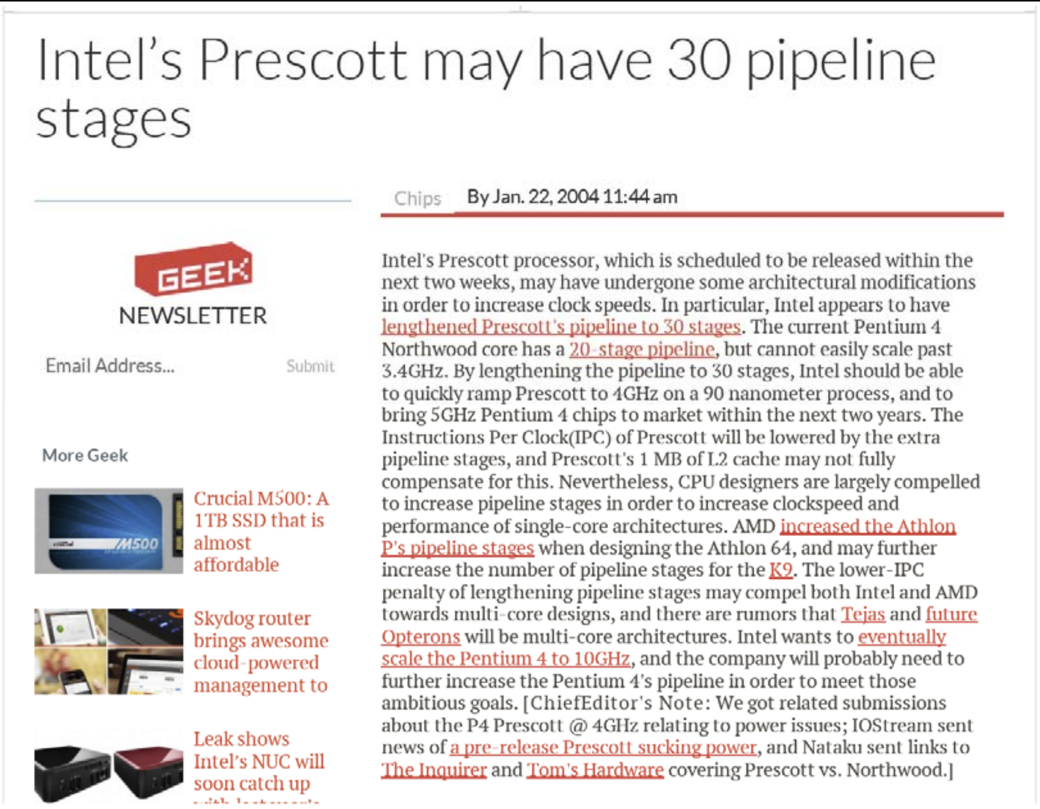
- An alternative to branch prediction is to execute both directions of a branch *speculatively*
  - resource requirement is proportional to the number of concurrent speculative executions
  - only half the resources engage in useful work when both directions of a branch are executed speculatively
  - branch prediction takes less resources than speculative execution of both paths
- *With accurate branch prediction, it is more cost effective to dedicate all resources to the predicted direction*

Does Multiple Issue Work?

The BIG Picture

- **Yes, but not as much as we'd like**
- **Programs have real dependencies that limit ILP**
- **Some dependencies are hard to eliminate**
  - **e.g., pointer aliasing**
- **Some parallelism is hard to expose**
  - **Limited window size during instruction issue**
- **Memory delays and limited bandwidth**
  - **Hard to keep pipelines full**
- **Speculation can help if done well**

10Ghz Processor?



Intel's Prescott may have 30 pipeline stages

Chips By Jan. 22, 2004 11:44 am

Intel's Prescott processor, which is scheduled to be released within the next two weeks, may have undergone some architectural modifications in order to increase clock speeds. In particular, Intel appears to have lengthened Prescott's pipeline to 30 stages. The current Pentium 4 Northwood core has a 20-stage pipeline, but cannot easily scale past 3.4GHz. By lengthening the pipeline to 30 stages, Intel should be able to quickly ramp Prescott to 4GHz on a 90 nanometer process, and to bring 5GHz Pentium 4 chips to market within the next two years. The Instructions Per Clock(IPC) of Prescott will be lowered by the extra pipeline stages, and Prescott's 1 MB of L2 cache may not fully compensate for this. Nevertheless, CPU designers are largely compelled to increase pipeline stages in order to increase clockspeed and performance of single-core architectures. AMD increased the Athlon P's pipeline stages when designing the Athlon 64, and may further increase the number of pipeline stages for the K9. The lower-IPC penalty of lengthening pipeline stages may compel both Intel and AMD towards multi-core designs, and there are rumors that Tejas and future Opterons will be multi-core architectures. Intel wants to eventually scale the Pentium 4 to 10GHz, and the company will probably need to further increase the Pentium 4's pipeline in order to meet those ambitious goals. [ChiefEditor's Note: We got related submissions about the P4 Prescott @ 4GHz relating to power issues; IOStream sent news of a pre-release Prescott sucking power, and Nataku sent links to The Inquirer and Tom's Hardware covering Prescott vs. Northwood.]

GEEK NEWSLETTER

Email Address... Submit

More Geek

Crucial M500: A 1TB SSD that is almost affordable

Skydog router brings awesome cloud-powered management to

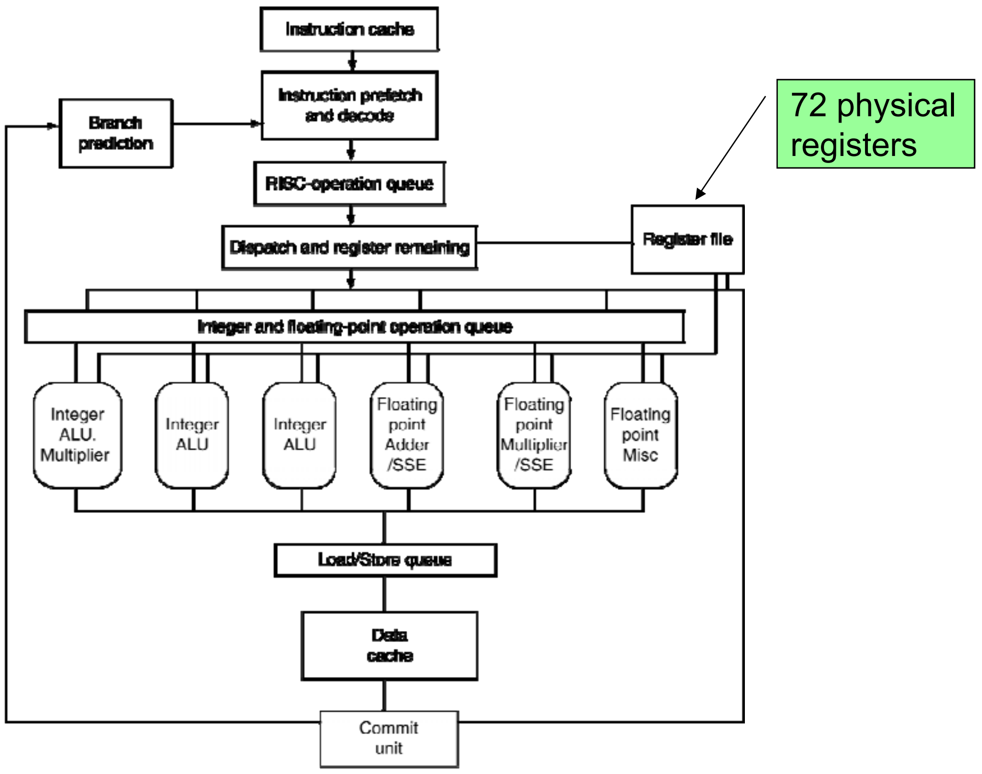
Leak shows Intel's NUC will soon catch up

Power Efficiency

- Complexity of dynamic scheduling and speculations requires power
- Multiple simpler cores may be better

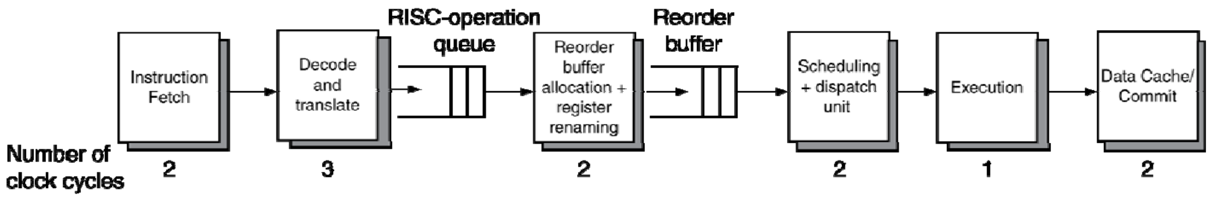
Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/ Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000MHz	22	3	Yes	1	75W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W
Core	2006	2930MHz	14	4	Yes	2	75W
UltraSparc III	2003	1950MHz	14	4	No	1	90W
UltraSparc T1	2005	1200MHz	6	1	No	8	70W

The Opteron X4 Microarchitecture



The Opteron X4 Pipeline Flow

- **For integer operations**



  - **FP is 5 stages longer**
  - **Up to 106 RISC-ops in progress**

- **Bottlenecks**
  - **Complex instructions with long dependencies**
  - **Branch mispredictions**
  - **Memory access delays**

Fallacies

- **Pipelining is easy (!)**
  - **The basic idea is easy**
  - **The devil is in the details**
    - e.g., detecting data hazards
- **Pipelining is independent of technology**
  - **So why haven't we always done pipelining?**
  - **More transistors make more advanced techniques feasible**
  - **Pipeline-related ISA design needs to take account of technology trends**
    - e.g., predicated instructions

Pitfalls

- **Poor ISA design can make pipelining harder**
  - **e.g., complex instruction sets (VAX, IA-32)**
    - Significant overhead to make pipelining work
    - IA-32 micro-op approach
  - **e.g., complex addressing modes**
    - Register update side effects, memory indirection
  - **e.g., delayed branches**
    - Advanced pipelines have long delay slots

Concluding Remarks

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control
- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall