The Evolution of High-Performance ISAs

Bruce Shriver

University of Tromsø, Norway

In The Early Days

- on-chip interconnections were fast and cheap; off-chip interconnections were slow and expensive
- registers were fast and expensive; main memory was slow and cheap;
- instructions took a long time to fetch relative to their decode and execution time, once fetched

Early Days (cont.)

- an early design goal: reduced the # of instructions and data fetched (reduced the overall computation time by reducing the aggregate fetch time)
- made powerful instructions (high vs low semantic content)
- densely encoded instructions reduced the execution time, but often had long decode times
- used instruction-level pipelining

Contending With An Inherently Serial Instruction

- example: M[x] + R[I] ➔M[x]
- example: M[y] + R[J] ➔M[z]
- *how* can we achieve performance improvements when such instructions must be executed?

Concurrent Execution

- parallelism can be achieved from the concurrent execution of instructions
- but pipelining does this, can it help?
- not really
- how else can we achieve the concurrent execution of instructions?

More Evolution

- in the late ‘70s & early ‘80s, high-performance memories became available at a reasonable cost
- performance was then limited by the decode and the execution times, not by the instruction fetch time
- <u>another design goal</u>: reduce the # of cycles required to decode and execute an instruction

Evolution (cont.)

- simple, regular instruction formats led to simple, efficient decoding
- minimize accesses to memory for operands by using a large number of *general-purpose* registers
- <u>yet another design goal</u>: the ISA should lend itself to pipelining

Evolution (cont.)

- in parallel with much of this history, other approaches were attempted, e.g multiple functional units were introduced
- recall this?: "*contending with an inherently serial instruction*
  - *parallelism can be achieved from the concurrent execution of several instructions*
  - *can pipelining help?*"
- can multiple functional units help?

Evolution (cont.)

Processors that dealt with instruction level parallelism by employing combinations of multiple functional units, high-performance memories, wide high-bandwidth buses, etc. were almost always considered to be "*on the bleeding edge*".

Bottom Line

- some ISAs are easier to pipeline than others
- pipelines are, by themselves, insufficient to be the basis of performance improvement; parallelism via multiple functional units must be correctly used
- <u>caches are fundamental</u>: hiding the time required for instruction & data fetches is critical
- <u>conflicting goals</u>: minimizing the number of instructions to get the job done and maximizing the number of instructions per cycle that get executed

Combining Forces

- use pipelining
- use parallelism
- technology constraints
  - process and packaging technology
  - memory and cache architecture and technology
  - ILP theory and practice
  - compiler technology
  - knowledge of the execution behavior of complex systems (e.g., the OS & various application domains)

Practical Issues Not On *Technology Constraints* List

- pinouts, gate counts, core logic and motherboard requirements, power, heat dissipation, etc.
- industry “standards”
- electronic design, simulation, and synthesis technology tools
- intellectual property related issues (applicable patents, copyrights, and trade secrets)

Now let’s take a deeper look into pipelines ...

Pipelines

- consist of a series of connected stages
- stages are connected by clocked latches
- items move from stage to stage, occupying only one stage at a time
- new items in the pipe follow older ones from stage to stage
- as an item leaves a stage, the item immediately following it enters the stage

Non-Pipelined vs Pipelined

(quote from Stone)

The basic assumption for programs written for a non-pipelined machine is that "*instructions are executed sequentially, with one instruction fully completed before the next is started*". A pipelined implementation "*violates this assumption, but it must give the appearance that the assumption holds*."

Pipelines (cont.)

- Stone: "… *the performance improvement depends on the ability to split the processes into steps of equal duration*."
- the slowest stage in the chain controls the movement through all other stages

Pipelines (cont.)

- do not reduce the total amount of time to execute an instruction
- reduce the average number of cycles (typically by the number of stages in the pipe)
- see Stone’s Figs. 3-1 and 3-2, Flynn’s Figure 2.9. and Johnson’s Figs. 1-1 and 1-2.

Pipelines (cont.)

- the memory architecture and technology must be able to support the processors rate of consuming instructions and data
- bank addressing, interleaving, bursting, etc. — matching speeds

Scalar Processors

- an instruction in a scalar processor executes on scalar data (as opposed to executing on vector or array quantities)
- scalar processors can and do employ pipelining
- when they do, they allow *one* instruction to execute in each pipeline stage; thus multiple instructions can be in execution at the same time *in different pipeline stages*

Superscalar Processors

- superscalar processors also operate on scalar data
- they also use pipelining; thus they can concurrently execute multiple instructions
- as with scalar processors they can have multiple instructions in execution at the same time *in different pipeline stages*
- but, they also allow concurrent multiple instruction execution *in the same pipeline stage* and it is this characteristic that gives them their name, superscalar
- there is thus the possibility to execute more that one instruction/cycle

More of Johnson’s Figures

- **1-3**: Pipelining in a Fetch-Limited (CISC) Processor
- **1-4**: A CISC Processor Without Memory Limitation
- **1-5**: A RISC Processor Without Memory Limitations
- **1-6**: Instruction Timing in a Superscalar Processor

Definitions and Notions (Kogge)

- pipelining & stage: "*Pipelining … generally takes the approach of splitting the function to be performed into smaller pieces and allocating separate hardware to each piece, termed a stage*."
- "… *instructions, or data, flow through the stages of a digital computer pipeline at a rate that is independent of the length of the pipeline (number of stages) and dependent only on the rate at which new entries may be fed to the input of the pipeline*."

More from Kogge

- "*This rate in turn depends primarily on the time for one piece of data to traverse a single stage*."
- overlap: "[the] *concurrent use of many different stages by different items*."
- "… *systems are often hierarchically designed with each stage for one level of pipelining itself actually designed as a pipeline*."

Kogge's *Hazards*

- hazard: "… *a hazard in a pipeline is basically an aspect of its design or use that prevents new data from continuously entering at the maximum possible rate*."
  - structural hazards: *"… where two different pieces of data attempt to use the same stage at the same time.*"
  - data-dependent hazards: *"… occur when what is going on in one stage of a pipeline determines whether or not data may pass through other stages*"

Pipeline Stalls

- interlock (Stone): a control device or signal that defers the execution of one function until a conflicting function has completed execution
- introducing delays in the pipeline
- wait states
- bubbles

Historical Examples of Systems

- early instruction level parallelism pipelining
  - STRETCH {IBM, 1959} - interleaved memory combined with a two-stage execution instruction pipe (a “instruction fetch/decode” phase and a “data execution phase”)
  - LARC {Eckert, 1959} - four-stage pipe (instruction fetch, address index operations, data fetch, and execution stages)
- early floating point operation pipelining
  - IBM 360/91 {1967}
  - TI’s ASC {1972}

Historical Examples (cont.)

- pipelining an instruction-set architecture not specifically tailored to a pipelined implementation, IBM 360/91 {1967}
  - much attention was paid to partitioning the system and to interlocks
  - general instruction execution pipe supported "*partial fetching of both possible sets of code following a conditional branch instruction when the results of the condition test depend on instructions yet complete. When the result is known, the unwanted sequence is purged.*"

More on the 360/91

- employed a hierarchy of pipelines, starting with the I-Unit and the E-Unit, both of which were pipelined
- the floating point units were "*connected by a decentralized common data bus, which initiates operations as soon as data operands are available, largely independent of the original order of instructions listed by the programmer*."
- show Figure 1-5 from Kogge's 1981 book, **The Architecture of Pipelined Computers**

Historical Examples (cont.)

- attempt to detect and reduce the impact of dependencies at the instruction level CDC 6000 {1964}
  - simple instruction formats
  - simple memory addressing (three register format)
  - use of many independent function units
  - use of a “central scoreboard” to check on dependencies and possibly deferring the execution in a stage

Historical Examples (cont.)

- early pipelining in vector processors
  - IBM 2938 {1969}
    - essentially a vector coprocessor (called an attached vector processor)
  - CDC STAR-100 {1972}
    - included both conventional and vector instructions
    - multiple independent pipelined functional units for the execution stage
    - bit vector addressing mode
  - CRAY-1 {1976}
    - more later

Historical Examples (cont.)

- early pipelining in array processors
  - IBM 3838 {1976}
    - a coprocessor type design
    - several levels of pipelining and overlap to be active
    - simultaneous interleaving of multiple programs for independent users
  - CRAY-1 {1976}
    - the vector instruction set applied to elements in vector registers
    - supported *chaining* of vector operations

Synchronous (or Static) Pipelining
A Strict Definition (Kogge)

"*Pipelining … refers to design techniques that introduce concurrency into a computer system by taking some basic function … and partitioning it into several sub-functions with the following properties:*

1. *Evaluation of the basic function is equivalent to some sequential evaluation of the subfunction*
2. *The inputs for one subfunction come totally from outputs of previous subfunctions in the evaluation sequence*
3. *Other than the exchange of inputs and outputs, there are no interrelationships between the subfunctions*
4. *Hardware may be developed to execute each subfunction*
5. *The times required for these hardware units to perform their individual evaluations are usually approximately equal.*"

A Few More Things

- unifunction and multifunction pipes
- static pipelining
- dynamic pipelining

SIMD/MIMD Parallel Machines
A Different Approach Altogether

- ILLIAC IV {1968} an array processor that also employed a pipelined and overlapped control-unit design
- C.mmp {1972} replicated minicomputers
- PEPE {1974}
- CM* {1977} replicated minicomputers
- TMC, KSR, HEP, etc. etc.

Pipelines vs Parallel Machines

- differences in memory organization
  - bandwidth (reads/sec & writes/sec) the memory must sustain: burst, quiescence vs. smooth
- differences in control
  - complexity of the interconnect (cost-performance) vs complexity of the pipes (cost-performance)
- differences in scalability

Ignoring History

- the insights gained in the design and implementation of such systems *should not be ignored* just because the graveside of failed firms is cluttered with scores of the fallen
- big machine bigots, CISC machine bigots, RISC machine bigots, …
- processors vs systems