Architecture as Interface

André DeHon
<andre@cs.caltech.edu>
Friday, June 21, 2002

Previously

- How do we build efficient, programmable machines
- How we mix
  - Computational complexity
  - W/ physical landscape
- To **engineer** computations
First 20 years…

• For the first 20 years of computing
  – That was all that mattered
• Built machine
  – Machines were scarce
  – Wrote software for that machine
• Each machine required new software
• OK, for small programs
  – Small uses of computation…

But…

• Technology advanced
  – Relays, tubes, discrete transistors, TTL, VLSI….
• The uses for automated computing grew
• We had a “Software Problem”
  – Couldn’t (re)write software fast enough
Questions

- How do we produce a growing amount of software for increasingly new generations of machines?
- How do we preserve software while exploiting new technology?

Ideas

- Define **model** abstracted from the details of the machine
- Write software to that model:
  1. Translate from model to machine (compiler)
  2. Preserve abstraction while changing technology (**architecture**)
Today

• **Architecture** – the visible interface between the software and the hardware
  – What the programmer (compiler) sees the machine as
  – The **model** of how the machine behaves
  – Abstracted from the details of how it may accomplish its task
  – What should be exposed vs. hidden?

Level

• Different level than first three lectures
  – Complexity management
  – Vs. existential and engineering
Today

- Architecture
- Examples
  - Nor2
  - Memory
  - Fault

Model: nor2 netlist

- Model of Computation:
  - Acyclic graph of nor2 gates
  - Repeat:
    • Receive input vector to the graph
    • Produce an output some time later
    • Signal completion
Represent

- Represent computation as a set of input pairs
  - Give each gate a number
  - For each gate, list its pair of inputs

\[
\begin{array}{cc}
  I_0 & I_2 \\
  I_1 & I_3 \\
  I_2 & I_3 \\
  1 & 2 \\
  2 & 3 \\
  4 & 5 \\
  4 & 3
\end{array}
\]

nor2 architecture

- This is the visible interface
  - List of gate inputs
- Maybe the hardware reads this directly
- But, hardware free to implement in any fashion

\[
\begin{array}{cc}
  I_0 & I_2 \\
  I_1 & I_3 \\
  I_2 & I_3 \\
  1 & 2 \\
  2 & 3 \\
  4 & 5 \\
  4 & 3
\end{array}
\]
Microarchitecture

- Microarchitecture – organization of the machine below the visible level
- Must provide same meaning (semantics) as the visible architecture model
- But can implement it any way that gets the job done

nor2-μarch: Temporal

- Temporal gate μarch satisfies arch:
nor2-µarch: Spatial

- As does our spatial array:

nor2-µarch: Superscalar

- Or even 2-gate, time-multiplexed:
Satisfying Architectures

- All provide the same meaning
  - Support the same abstraction
  - Produce the same visible results
- Abstract away
  - Timing
    - done in 20 nanoseconds
    - done in 2 milliseconds
  - Internal structure (placement, parallelism)

Traditional

- Traditional “architecture” model
  - ISA – Instruction Set Architecture (e.g. x86)
  - **Model**: sequential evaluation of instructions
  - Define meaning of the instructions
Arch vs. µarch

- Model says:
  - issue one instruction at a time
- Modern machines:
  - Issue more than one instruction at a time
    - Like “superscalar” nor2
    - Superscalar = faster than single issue
  - But preserves the semantics of sequential issue

Benefit

- Preserve our software investment
  - Same programs continue to run
- While exploiting more and faster hardware which technology provides
- More parallelism over time
Note

• Conventional model is of a sequential machine
  – Model serves to limit parallelism, use of more hardware
• nor2 model was of a parallel evaluation
  – Parallelism exposed
  – Admits to large-hardware implementation
  – Perhaps harder to optimize for small/sequential implementations…
Memory Abstraction

• Give it: operation, address, data(?)
• Changes data at address (if write)
• Gives back data at address (if read)
• Allow memory to signal when done
  – Abstract out timing

Memory Abstraction

• Memory abstraction simple
• But we do many things “behind the scenes” to optimize
  – Size/speed
  – Organization
  – failure
Issue: Memory Size

• Larger memories are slower:
  – Simple physics:
    • In M-bit memory
    • Some bits are $\sqrt{M}$ distance from interface
      – In fact most are that far
    • Speed of light limits travel speed
    • As $M$ increases, memory delay increases
  
• But, we want to address more stuff
  – Solve larger problems

Memory Acceleration Idea

• Since:
  – Small memories are fast
  – Large memories are slow
  – Need access to large amount of data

• Use both:
  – Small memory to hold some things
    • Responds quickly
  – Large memory to hold the rest
    • Response more slowly
Memory Acceleration

• Exploit Model
  – Signal when operation complete
  – Access to small memory returns quickly
  – Access to large memory returns more slowly
  – Always get correct result

To the extent commonly accessed stuff in fast memory
→ runs faster
Cache \( \mu \text{arch} \)

- More complicated implementation
- But implementation details invisible to program
  - Size of small/large memory
  - Access speeds

Memory Optimization

- May do this several levels
  - Modern machines have 3—4 levels of memory
- Use this to pretend have more memory
  - \textbf{Virtual memory} -- use disk as largest “memory”
- Can get very complicated
  - But offers a single, simple, visible \textbf{architecture}
Issue: Perfection

- Large memories $\Rightarrow$ more likely to have faulty components
  - Simple probability
    - Assume some probability of a defect per memory cell
    - Increase memory cells
    - Probability all are functional?
- But we want larger memories
  - And our model says the device is perfect

General Idea

- Provide extra memories
- “Program” device to substitute good resources for bad
- Export device which meets model
  - Appears perfect
Row Redundancy

• Provide extra memory rows
• Mask faults by avoiding bad rows
  – Rows with bad bits
• Trick:
  – have address decoder substitute spare rows in for faulty rows
    • use fuses to program

Spare Row
Memory Arch vs. μarch

- Microarchitecture more complicated
- Changes from implementation to implementation
  - Number of spares
    - Increase with device size
    - Decrease with process maturity
  - Invisible to architecture
Issue: Dynamic Faults

- What happens if a bit changes dynamically?
  - Hit by an alpha particle?
    - Deplete charge in cell
    - Memory gives wrong answer
    - Not providing model

Fault Approach

- Keep redundant information in memory
- Simplest case: keep multiple copies
  - Read values and take majority
  - Return majority from memory
  - Probability of m-simultaneous faults low
  - Pick redundancy appropriately
Redundant Memory

addr

• Model remains simple
  – Perfect memory
• Microarchitecture complex
  – Write multiple copies
  – Compare copies
  – Correct
• Microarchitecture addresses/hides physical challenges
  – Non-zero probability of bit faults

Memory Arch vs. µarch
Better Encodings

- In practice, don’t have to keep many copies
- Encode many bits sparsely
  - *E.g.* use 5 bit codes for 4 bits
  - *E.g.* parity code
    - Detect single bit errors
  - Better: error correcting codes which will
    - Correct single bit errors
    - Detect many multi-bit errors
    - *E.g.* conventional memory systems 64b in 72b

Alternate Fault Model
Two Common Defect Models

- Memory Model (just discussed)
- Disk Drive Model

Memory Chips

- Provide model in hardware of perfect component
- Model of perfect memory at capacity X
- Use redundancy in hardware to provide perfect model
- Yielded capacity fixed
  – discard part if not achieve rated capacity
Disk Drives

- Expose faults to software
  - software model expects faults
  - manages by masking out in software
    - Don’t expect to be able to use all sectors (bytes)
    - OS only allows you to to access the good ones
    - Bad ones look like they are in use by someone else
  - yielded capacity varies with defects

Wrapping Up
“Architecture”

- “attributes of a system as seen by the programmer”
- “conceptual structure and functional behavior”
- Defines the visible interface between the hardware and software
- Defines the semantics of the program (machine code)

Architectural Abstraction

- Define the fixed points
- Stable abstraction to programmer
- Admit to variety of implementation
- Ease adoption/exploitation of new hardware
- Reduce human effort
- Reduces the “software problem”
Enables

- Allow devices to scale
- Allow software to survive across and benefit from device scaling
- Allow us to optimize implementations
- Allow us to hide physical challenges from software