Homework Slides - 1/28/2013

Jaylen VanOrden

S1 - Algorithm for Multiple ALUs

- Floating-point system in IBM System/360 Model 91
- Goals:
  - Use several ALUs simultaneously while preserving instruction ordering
  - No specialized code
- Observations:
  - Generalized modules are slower
  - Split floating point vs. fixed point
  - Split adder vs. multiplier (more difficult)
- Data dependency illustrated by 'load, then multiply' ex.
  - System must recognize dependency,
  - keep correct sequence, and
  - allow independent instructions to be parallelized

S2 - Algorithm for Multiple ALUs

- 'Busy bit' enforces precedence, but doesn't encourage overlap of independent operations
- 'Reservation stations' allow limited queuing(?) of operations and some overlap, but still needs programmer help
- Another problem - 2 kinds of operation independence:
  - Second set of ops has different sink registers
  - Second set of ops starts with a load (loop example)
- Common data bus
  - Connects sources and sinks of fp. data
  - Tags allow data coordination + overlap of execution
- Result: about ~1/3 speedup (in PDE calculation)

?s - Algorithm for Multiple ALUs

- The tag dependency system in the CDB seems to be the 'magic' that allows it to handle concurrent operations while maintaining dependencies. However, each tag is broadcast to all units, which seems like a future (post-1967) bottleneck. How well does this concept scale?
- How do modern processors handle this problem, especially when the number of ALUs and other execution centers is much larger?

S1 - HPSm, Minimal Data Flow Arch.

- Goals:
  - High concurrency in hardware
  - Use this concurrency well (minimize stalls)
- Idea: use data flow to parallelize sequential code exec.
  - Focusing on local parallelism
  - Use an 'active window' of current ops to execute
  - Node table acts as buffer for larger amounts of data dependencies
- HPS model
  - Decoder: converts each instruction to a data dependency graph
  - Merger: merges each new dependency graph into the graph for the active window
  - Functional units compute when operands are ready

S2 - HPSm, Minimal Data Flow Arch.

- Memory writes are finalized only when all previous instructions and the writing one have executed
- 3 types of dependencies to handle:
  - Flow - op1 writes to a register op2 reads
  - Anti - op2 will write to a register op1 reads
  - Output - op1 and op2 will write to the same register
- We could avoid anti and output dependencies if we used temporary registers, since op2 doesn't depend on op1's output in these cases
- Two types of repair are necessary:
  - Branch prediction misses (common)
  - Instruction exceptions (rare)
  - Both can be handled by storing machine state

S3 - HPSm, Minimal Data Flow Arch.

- HPSm
  - 32-bit opcodes
  - 2 ops per instruction
  - Pass-by-register convention for parameters
  - Limited branch prediction & HW parallelism
  - Cache for decoded instructions
  - Register alias table stores backup on branch pred.
  - All operations have 2 inputs
  - Tested using hand-translated/optimized assembly
- Tested versus RISC II processor, initial results were very strong.

?s - HPSm, Minimal Data Flow Arch.

- By the end of this paper, I noticed that the 'efficient algorithm' paper looks alot like data flow and, in fact, was in the references for the HPSm paper. When was the 'data flow' term coined? Is data flow simply an extension of parallelizing multi-ALU architectures?
- Did HPSm implement multiplication/division? If not, how much effect did this simplification have on the processor speed?