Computer Architecture Review

Mark Greenstreet

CpSc 418 – September 28, 2018

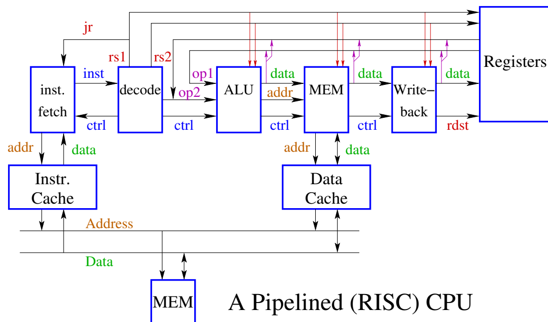
- A microcoded machine
- A pipelined machine: RISC
- Superscalars and the memory bottleneck

Unless otherwise noted or cited, these slides are copyright 2018 by Mark Greenstreet and are made available under the terms of the Creative Commons Attribution 4.0 International license
http://creativecommons.org/licenses/by/4.0/

Objectives

- Pipelining review
  - Pipelining **is** parallel execution.
  - The machine is supposed to appear (nearly) sequential.
  - Tntroduce the ideas of hazards and dependencies.

Pipelined instruction execution



A Pipelined (RISC) CPU

- Successive instructions in each stage
- When instruction `i` in `ifetch`, instruction `i-1` in decode, ...
- Allows throughput of one instruction per cycle.
- Favors simple instructions that execute on a single pass through the pipeline.
  - This is known as RISC: “Reduced Instruction Set Computer”
  - A modern x86 is CISC on the outside, but RISC on the inside.

What about Dependencies?

- Multiple-instructions are in the pipeline at the same time.
- An instruction starts before all of its predecessors have completed.
- Data hazards occur if
  - an instruction can read a different value than would have been read with a sequential execution of instructions,
  - or if a register or memory location is left holding a different value than it would have had in a sequential execution.
- Control hazards occurs if
  - an instruction is executed that would not have been executed in a sequential execution.
  - This is because the instruction "depends" on a jump or branch that hasn't finished in time.

Handling Hazards

- Bypass: If an instruction has a result that a later instruction needs, the earlier instruction can provide that result directly without waiting to go through the register file.
- Move common operations early:
  - Decide branches in decode stage
  - ALU operations in the stage after decode
  - Memory reads take longer, but they happen less often.
- Let the compiler deal with it
- If nothing else helps, stall.

RISC summary

- without pipelining, a CPU typically takes $5+$ clock-cycles per instruction.
- the RISC machine takes 1 clock-cycle per instruction – in the best case:
  - There can be stalls due to cache misses,
  - unfilled delay slots, or
  - multi-cycle operations.
- Can we break the one-cycle-per instruction barrier?

Rules for executing instructions

- A programmer understands a process or thread as executing a sequence instructions.
  - The ordering of instructions in this sequence is called **program order**
  - The pgrogram should execute **as if** instructions are executed in program order.
- Hardware can get better performance by re-ordering instructions and executing instructions in parallel.
- The hardware must observe **dependencies**

Dependencies

- Let's say that we have instructions $inst_i$ and $inst_j$ where i and j are there indices in program-order execution, and $i <$.
- We say that instruction j **depends** on instruction i if
  - Read-After-Write (RAW): $inst_j$ reads a register or memory location written by $inst_i$. We need $inst_j$ to read the right value and not get a stale value.
  - Write-After-Read (WAR): $inst_j$ writes a location read by $inst_i$. We need $inst_i$ to read the right value and not get a value "from the future."
  - Write-After-Write (WAR): $inst_i$ and $inst_j$ write the same location. $inst_j$ must be the last write so that subsequent instructions see the correct value.
  - Control dependencies: $inst_i$ is a control flow instruction, for example, a branch, jump, function call, or trap. We need to execute the correct instructions.
- To ensure correct execution, if $inst_j$ depends on $inst_i$, then $inst_j$ must execute after $inst_j$.
  - **Unless** we can apply some technique for removing the dependency. We'll cover this in subsequent slides.

Matrix Multiplication and Performance

```
for(int i = 0; i < M; i++)
  for(int j = 0; j < N; j++)
    for(int k = 0; k < L; k++)
      c[i,j] += a[i,k]*b[k,j];
```

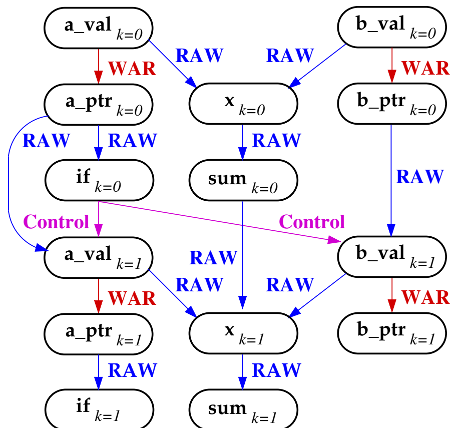
- Focus on the innermost loop: `for(k ...)`
  - Why?

The Inner Loop: Dot Product

- C
```
for(int k = 0; k < L; k++)
  c[i,j] += a[i,k]*b[k,j];
```
- Assembly (sketch):
```
LOOP:
  a_val ← *a_ptr              # load
  a_ptr ← a_ptr + a_stride    # a_stride = sizeof(double)
  b_val ← *b_ptr              # load
  b_ptr ← b_ptr + b_stride    # b_stride = N*sizeof(double)
  x ← a_val * b_val           # floating point multiply
  sum ← sum + x               # floating point add
  if a_ptr < a_max goto LOOP  # conditional branch
```

Dot-Product, The Dependency Graph

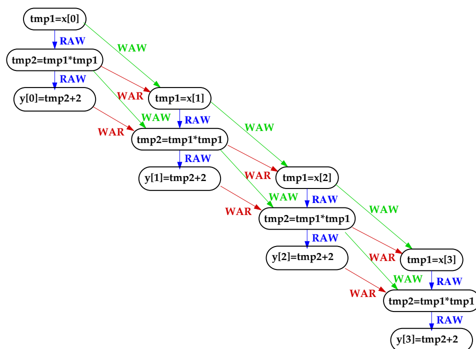
```
k=0:
  a_val <– *a_ptr
  a_ptr <– a_ptr + a_stride
  b_val <– *b_ptr
  b_ptr <– b_ptr + b_stride
  x <– a_val * b_val
  sum <– sum + x
  if a_ptr < a_max

k=1:
  a_val <– *a_ptr
  a_ptr <– a_ptr + a_stride
  b_val <– *b_ptr
  b_ptr <– b_ptr + b_stride
  x <– a_val * b_val
  sum <– sum + x
  if a_ptr < a_max
```



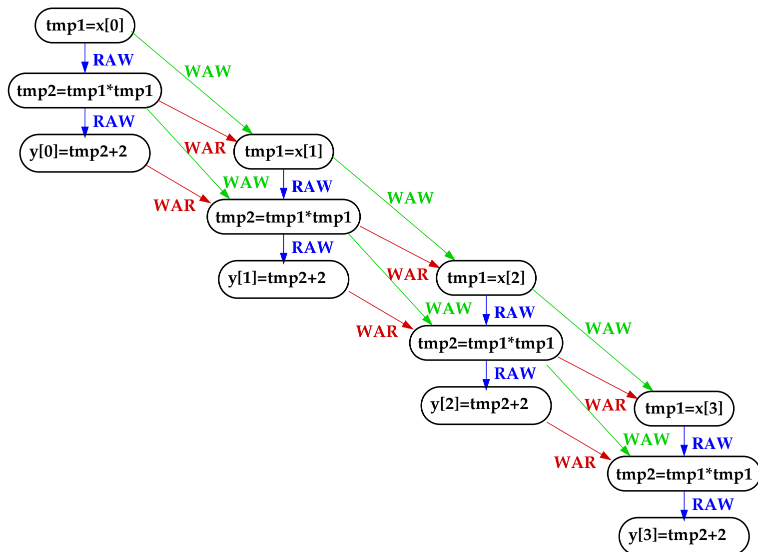
- An instruction can execute as soon as all instructions that it depends on have completed.
- This allows us to execute the 7 instruction loop in 3 cycles (with very idealized hardware).

$x^2 + 2$: Is it parallel?

```
% y[i] ← x[i]^2 + 2, for i ∈ [0..3]
tmp1 = x[0];
tmp2 = tmp1*tmp1;
y[0] = tmp2+1;
tmp1 = x[1];
tmp2 = tmp1*tmp1;
y[1] = tmp2+1;
tmp1 = x[2];
tmp2 = tmp1*tmp1;
y[2] = tmp2+1;
tmp1 = x[3];
tmp2 = tmp1*tmp1;
y[3] = tmp2+1;
```



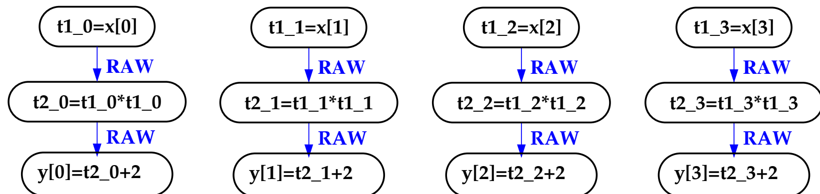
The Dependency Graph (zoom)



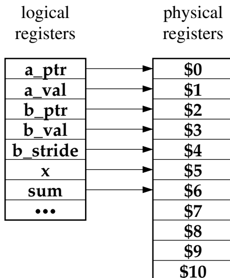
Reusing `tmp1` and `tmp2` is creating lots of WAR and WAW dependencies.

False Dependencies

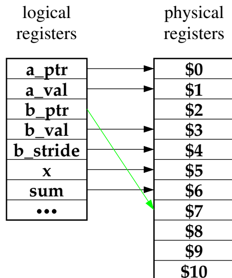
- WAR: If we had a fresh variable (register in the hardware), we could have the write go to a new register and not interfere with the earlier (in program order) read.
- Likewise for WAW.



Register Renaming



Before Decoding



After Decoding

- When an instruction is decoded
  - The logical registers that it reads are bound to physical registers according to the current register mapping.
  - A fresh register is allocated for the register it writes. This register is marked as busy.
- When an instruction updates its destination register, it changes it from "busy" to "ready".
- An instruction can execute when all registers that it reads are "ready".

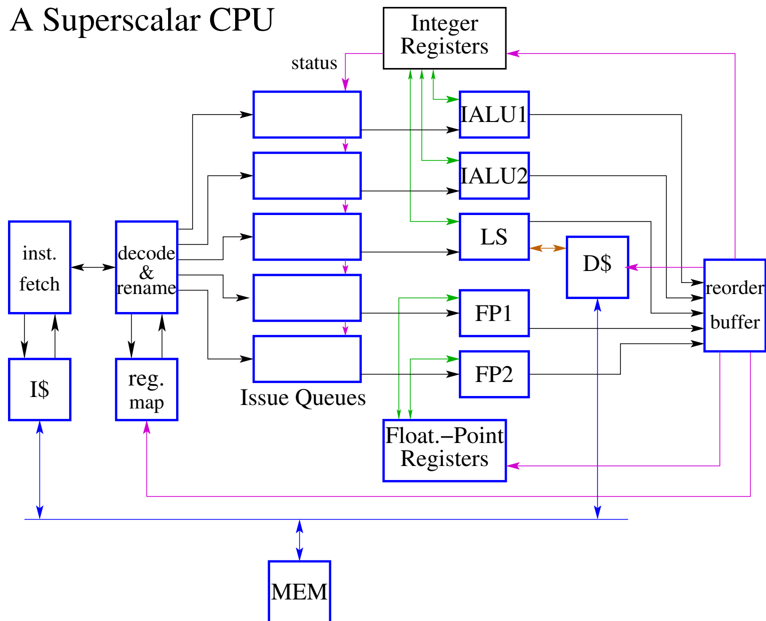
Control Dependencies

- If we use register renaming with our matrix-multiply (dot-product) example:
  - How many clock cycles per iteration?
- Now, the bottleneck is the control dependency for the for-loop branch.
  - For large matrices and/or vectors, the branch is taken "most of the time"
- Speculative execution
  - Track statistics for branch outcomes.
  - Speculatively execute the more likely path.
  - Roll-back if wrong.
    - Reset program counter.
    - Unwind register mappings back to the point of the mispredicted branch.

Other stuff

- Exceptions
- Committing instructions and freeing registerrs

Superscalar Processors



Superscalar Execution

- Fetch several, W, instructions each cycle.
- Decode them in parallel, and send them to issue queues for the appropriate functional unit.
- But what about dependencies?
  - We need to make sure that data and control dependencies are properly observed.
  - Code should execute on a superscalar as if it were executing on sequential, one-instruction-at-a-time machine.
  - Data dependencies can be handled by "**register renaming**" – this uses register indices to dynamically create the dependency graph as the program runs.
  - Control dependencies can be handled by "**branch speculation**" – guess the branch outcome, and rollback if wrong.
- The opportunity to execute instructions in parallel is called **Instruction Level Parallelism**, ILP.

What superscalars are good at

- Scientific computing:
  - often successive loop iterations are independent
  - the superscalar **pipelines** the loop
  - Perform memory reads for loop i, while doing multiplications for loop i-2, while doing additions for loop i-4, while storing the results for loop i-5.
- Commercial computing (databases, webservers, ...)
  - often have large data sets and high cache miss rates.
  - the superscalar can find executable instructions after a cache miss.
  - if it encounters more misses, the CPU benefits from **pipelined** memory accesses.
- Burning lots of power
  - many operations in a superscalar require hardware that grows quadratically with W.
  - basically, all instructions in a batch of W have to compare there register indices with all of the other ones.

Superscalar Reality

- Most general purpose CPUs (x86, Arm, Power, SPARC) are superscalar.
- Register renaming works **very** well:
- Branch prediction is also very good, often $>$ 90% accuracy.
    - But, data dependent branches can cause very poor performance.
- Superscalar designs make multi-threading possible
    - The features for executing multiple instruction in parallel work well for mixing instructions from several threads or processes – this is called “multithreading” (or “hyperthreading”, if you’re from Intel).
    - In practice, superscalars are often *better at multithreading* than they are at extracting ILP from a sequential program.

Lynn Conway

- Worked on first super-scalar processor design at IBM.
- Was subsequently fired from IBM. Why?
- Worked as a programmer at Memorex for a few years, and then went to Xerox PARC.
- Collaborated with Carver Mead (Caltech) to start the “VLSI revolution”
  - Key idea was to apply principles of abstract from computer science to integrated circuit design.
  - approach has completely transformed the industry and made large, multi-billion transistor designs possible.

- For more information, see:
  http://ai.eecs.umich.edu/people/conway/
  Photo from http://en.wikipedia.org/wiki/File:Lynn_Conway_July_2

Why does it matter?

- Role models matter
  - If your a straight, white or asian, male in computer science,
    - then there are lots of people like you who can be role models.
    - If you're like me, you'll often take this for granted, and not even think of them as role models.
  - The further you are from this "center-of-mass" of the field,
    - the sparser role models become,
    - and you may feel like you don't fit in.
    - If so, please get the message from this that you're not the problem.
- Lynn Conway has lived a remarkable life
  - She's made important contributions to computer architecture, VLSI design, and robotics.
  - She's chosen to use her experiences to help others who are facing similar challenges and discrimination.

Review

- How does a pipelined architecture execute instruction in parallel?
- What are hazards?
- What are dependencies?
- What is multithreading.
- For further reading on RISC:
  "Instruction Sets and Beyond: Computers, Complexity, and Controversy"
  R.P. Colwell, *et al.*, *IEEE Computer*, vol. 18, no. 3,
  - You can download the paper for free if your machine is on the UBC network.
  - If you are off-campus, you can use the library's proxy.