Superscalar Architectures

Mark Greenstreet

CpSc 418 – Oct. 1, 2013

Objectives:

- Understand how modern processors execute programs.
- See dependencies in the context of instruction execution.
- Better understand issues underlying performance.

Matrix multiply: from Sept. 26 lecture

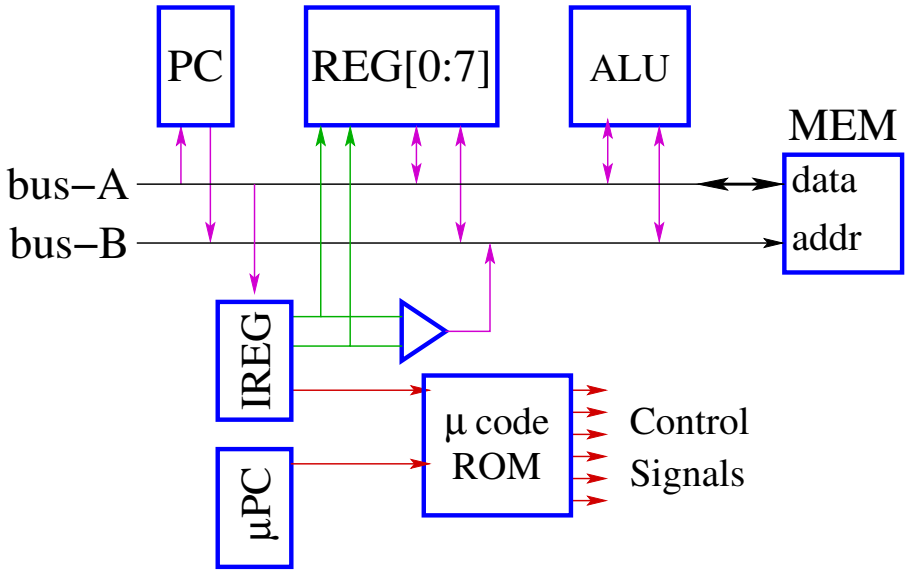
In C:

```
for(i = 0; i < n; i++) {
    for(j = 0; j < n; j++) {
        sum = 0.0;
        for(k = 0; k < n; k)
            sum += a[i,k]*b[k,j];
        c[i,j] = sum;
    }
}
```

Machine-level operations, just the inner-loop:

```
LOOP_TOP:
    $x ← Mem($aptr).double
    $aptr ← $aptr + 8
    $y ← Mem($bptr).double
    $bptr ← $bptr + $N8
    $z ← $x * $y
    $sum ← $sum + $z
    branch $aptr ≠ $atop, LOOP_TOP
```

Microcoded machines



A simple, microcoded machine

- The microcode (μcode) ROM specifies the sequence of operations necessary to carry out an instruction.
- For simplicity, I'm assuming that the op-code bits of the instruction form the most significant bits of the μcode ROM address, and that the value of the micro-PC (μPC) form the lower half of the address.

Matrix multiply on a microcoded machine

`$x ← Mem($aptr).double`: 5+ cycles:

1. Fetch instruction
2. Decode instruction
3. Fetch register, `$aptr`
4. Read memory (may take more than one cycle)
5. Write result into register $x

`$aptr ← $aptr + 8`: 5 cycles

1-3, 5: Like the load operation above.

4: An ALU operation instead of a memory operation.

`$z ← $x * $y`: 6 cycles

1-3, 5: Like the add operation above.

4: Assuming 2-cycle latency for floating point operations. See Table 2 in the *MIPS R10000* paper. Today's processors tend to have higher latencies, typically 3-5 cycles.

Matrix multiply on a microcoded machine (cont).

`branch $aptr ≠ $atop, LOOP_TOP`: Five cycles.

1-4 Like an ALU operation.

5 Update the program counter instead of a data register.

TOTAL: **36 cycles**.

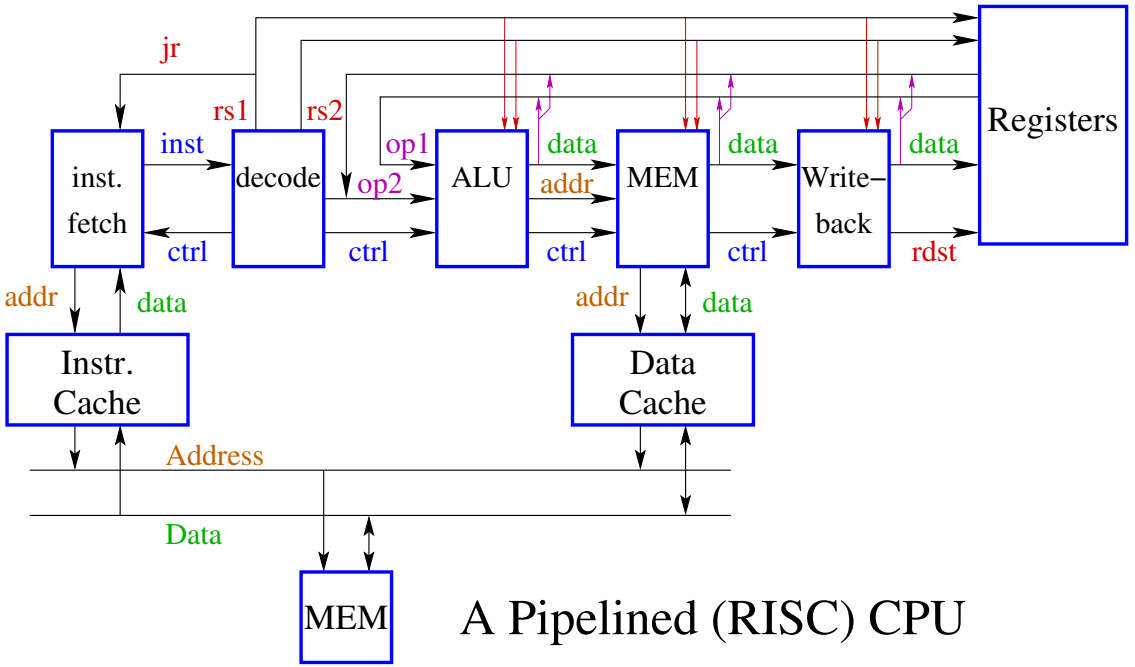
Microcode: summary

- Separates hardware from instruction set.
  - Different hardware can run the same software.
  - Enabled IBM to sell machines with a wide range of performance that were all compatible
    - I.e. IBM built an empire and made a fortune on the IBM 360 and its successors.
    - Intel has done the same with the x86.
- But, as implemented on slide 3, it's very sequential.

```
while(true) {
    fetch an instruction;
    perform the instruction
}
```

- Instruction fetch is "overhead"
  - Motivates coming up with complicated instructions that perform lots of operations per instruction fetch.
  - But these are hard for compilers to use.
  - Can we do better?

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.

What about Dependencies?

Multiple-instructions are in the pipeline at the same time.
An instruction starts before all of its predecessors have completed.

RAW: Instructions in the pipeline see the register reads of subsequent instructions.

- If the value is available, the register file is “bypassed.”
- Otherwise, the instruction in the decode stage stalls.

WAR: Instructions perform their writes at the very end, after all previous instructions have completed their reads.

WAW: Instructions update the register file (and memory) in program order.

Control:

- Branches are executed in the decode stage.
- The instruction after the branch is always fetched.
- Two choices:
  - Sqash that instruction if the branch is taken.
  - Execute it anyway – this is the “delay slot” approach.

Matrix multiplication on a pipelined machine

```
LOOP_TOP:
    $x ← Mem($aptr).double
    $aptr ← $aptr + 8
    $y ← Mem($bptr).double
    $bptr ← $bptr + $N8
    $z ← $x * $y
    $sum ← $sum + $z
    branch $aptr ≠ $atop, LOOP_TOP
```

- We can move the `$sum ← $sum + $z` operation in to the branch delay slot.
- This also prevents the floating point add from stalling while waiting for the floating point multiply to finish.
- The loop executes in **7 cycles** per iteration.
- But what if we have a modern processor with longer operation latencies?

Matrix multiplication on a pipelined machine

```
LOOP_TOP:
    $x ← Mem($aptr).double
    $aptr ← $aptr + 8
    $y ← Mem($bptr).double
    $bptr ← $bptr + $N8
    $z ← $x * $y
    $sum ← $sum + $z
    branch $aptr ≠ $atop, LOOP_TOP
```

- But what if we have a modern processor with longer operation latencies?
  - E.g., an Intel Core i7 has a three cycle latency for floating point add, and a five-cycle latency for floating point multiplication.
  - It can issue a new multiply and add every cycle.

Loop Pipelining

Make sure `n % 3 == 2`, and do some work before the main loop to get everything set-up

```
LOOP_TOP:
    $x0 ← Mem($aptr).double
    $z2 ← $x2 * $y2
    $y0 ← Mem($bptr).double
    $sum ← $sum + $z1
    $x1 ← Mem($aptr, 8).double
    $z0 ← $x0 * $y0
    $y1 ← Mem($bptr, $N8).double
    $sum ← $sum + $z2
    $x2 ← Mem($aptr, 16).double
    $z1 ← $x1 * $y1
    $y2 ← Mem($bptr, $N16).double
    $sum ← $sum + $z0
    $aptr ← $aptr + 24
    $bptr ← $bptr + $N24
    branch $aptr ≠ $atop, LOOP_TOP
```

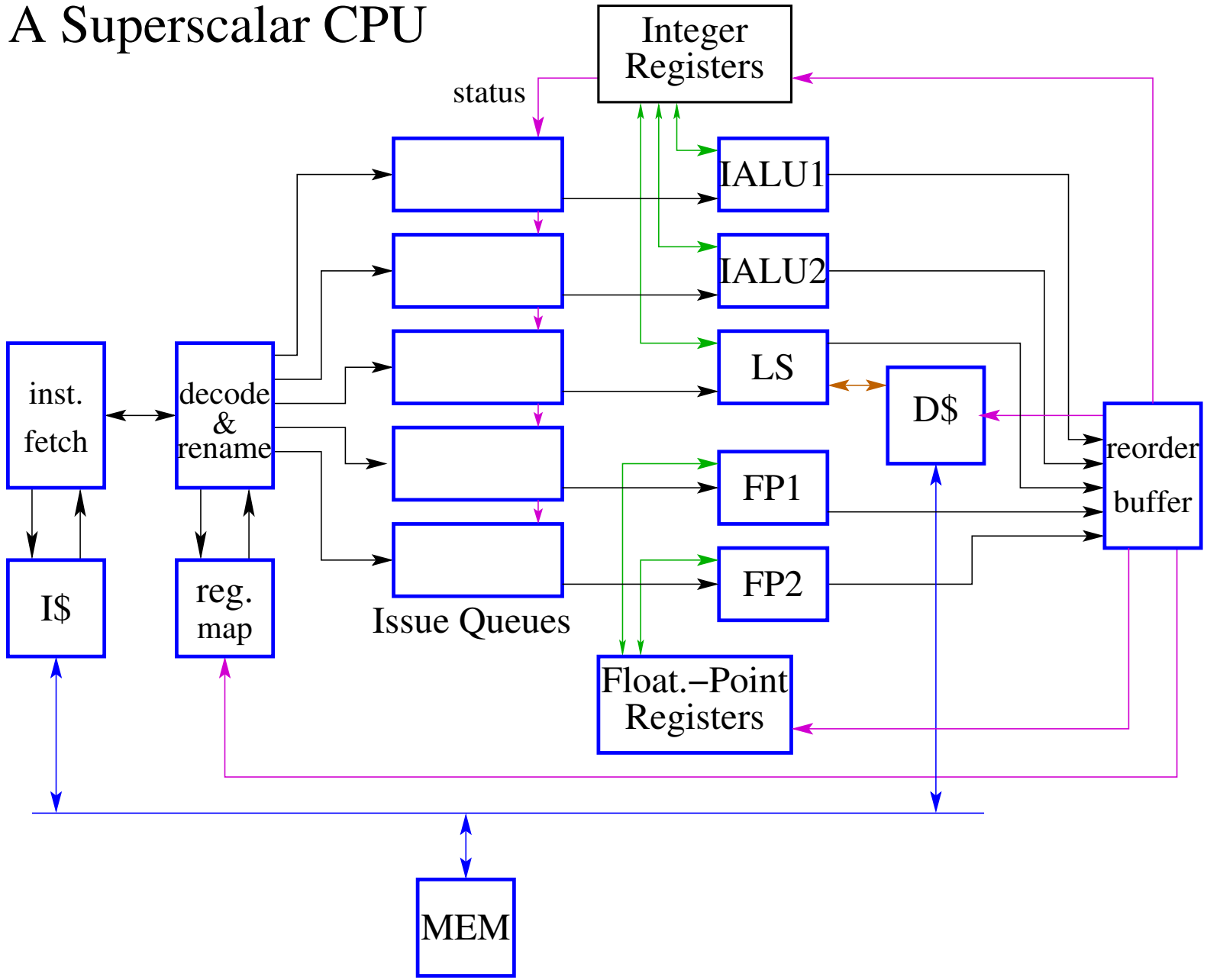
The rest of the loop

- Stuff to handle a few iterations in case `n % 3` $\neq$ `2`,
- Get the first two iterations started:

```
$x1 ← Mem($aptr).double
$aptr ← $aptr + 16
$y1 ← Mem($bptr).double
$bptr ← $aptr + 16
$x2 ← Mem($aptr, -8)
$z1 ← $x1 * $y1
$y2 ← Mem($bptr, -$N8)
```

- There will be a loop epilogue as well to finish the the last two iterations.
- The new code takes 15 cycles to perform 3 iterations of the original loop.
- **5 cycles per iteration!**

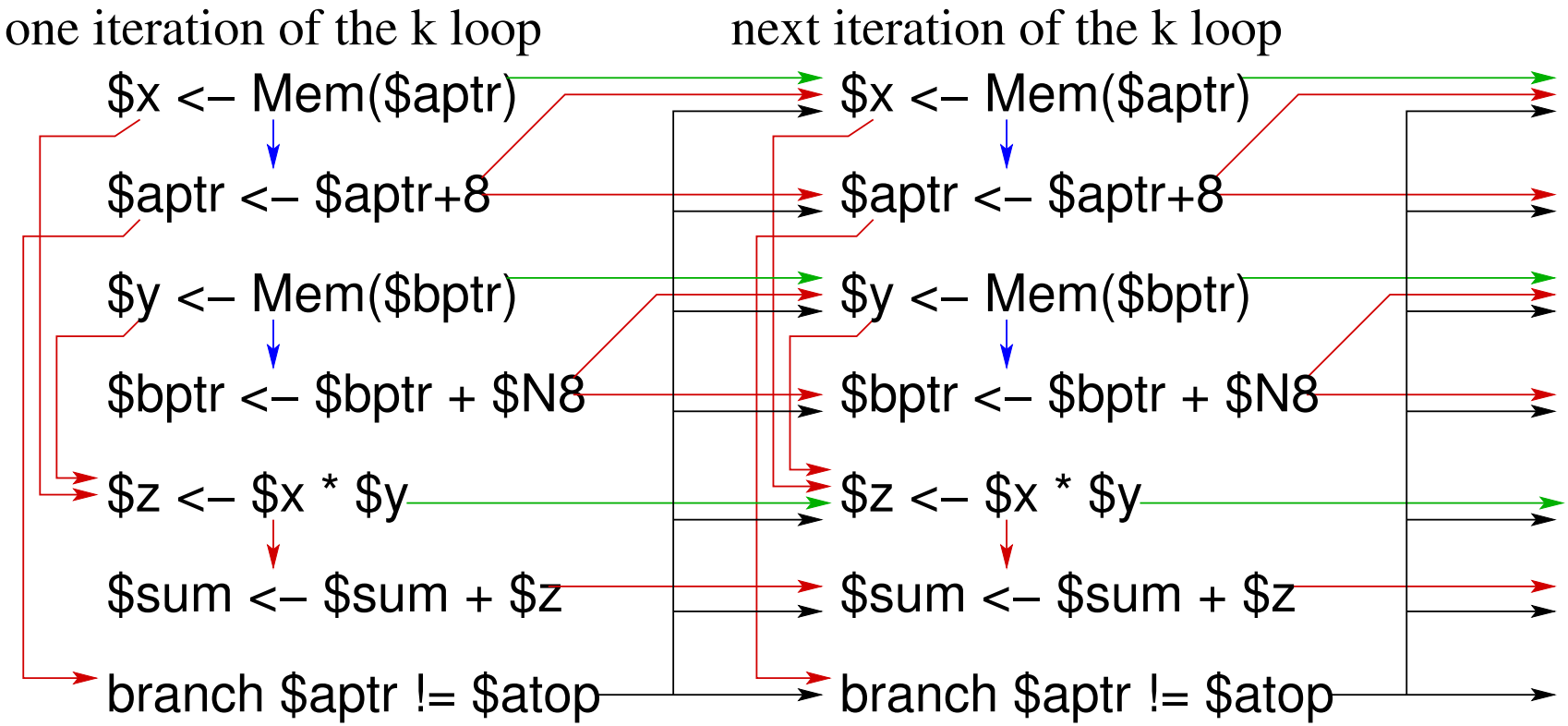
Superscalar Processors



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.

Dependencies



Dependencies: RAW, WAR, WAW, Control

Data Dependencies: Register Renaming

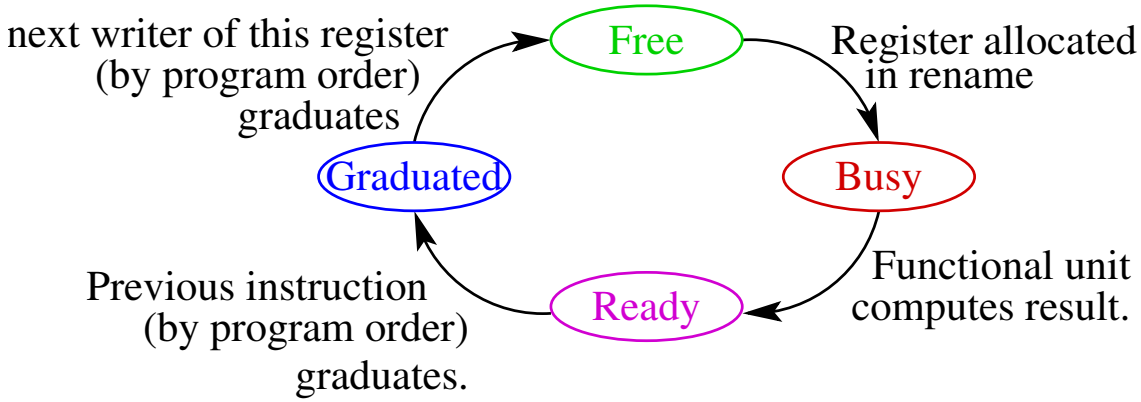
Like so much else in computer science, add a level of indirection.

- Logical and Physical Registers
  - Machine-code (assembly) instructions refer to logical registers.
  - The machine stores values in physical registers.
  - The machine maintains a mapping between the logical and physical registers.
- Renaming:
  - When an instruction is decoded,
    - logical registers that it reads are mapped to physical register according to the current register map.
    - for each register that the instruction writes:
      - a physical register is allocated from a free list.
      - the register map is updated to make this new physical register for the logical register written by the instruction.
  - I'll explain how registers get back to the freelist shortly.

Renaming Example

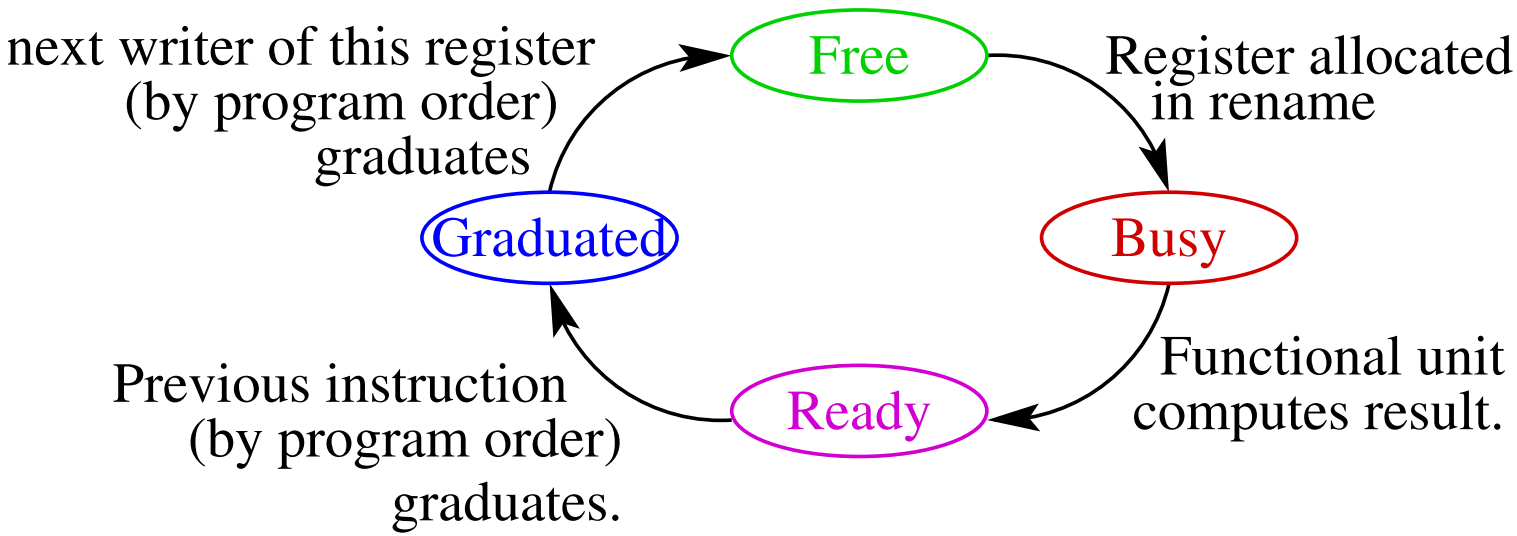
```
LOOP_TOP:
    $x_i ← Mem($aptr_{i-1}).double
    $aptr_i ← $aptr_{i-1} + 8
    $y_i ← Mem($bptr_{i-1}).double
    $bptr_i ← $bptr_{i-1} + $N8
    $z_i ← $x_i * $y_i
    $sum_i ← $sum_{i-1} + $z_i
    branch $aptr_i ≠ $atop, LOOP_TOP
```

The Lifecyle of a Register (1/2)



- Renaming constructs a graph that reflects the data dependencies of the program.
- It removes “false” hazards such as
  - Write-after-write:
    - Two instructions that write the same logical register can execute in either order.
    - The last one in program order will be the last one to graduate.
  - Write-after-read:
    - An instruction that writes a logical register can execute before an earlier instruction that reads the same logical register.
    - That’s because the two occurrences of the logical register map to different physical registers.

The Lifecyle of a Register (1/2)



- Renaming removes “false” hazards such as
  - Write-after-write.
  - Write-after-read.
- The set of graduated registers corresponds to the state of a sequential execution of the program after executing the last graduated instruction.

Branch Prediction

- A branch may not be resolved until several cycles after it is fetched.
- Many instructions will have been fetched, decoded, queued and perhaps executed (but not graduated) in the meantime.
  - How does the processor determine which instructions to fetch?
- Speculate
  - Keep track of recent history of branches – typically with "saturating counters'.
  - If the counter predicts branch taken, the fetch unit sees this and fetches from the branch target.
  - If the counter predicts branch not-taken, the fetch unit continues fetching in sequence.
  - If a branch is mispredicted
    - Then the wrong path instructions are aborted.
    - The processor resumes execution down the correct path.
    - Branch mispredict penalties can be large.

Does Branch Prediction Work?

Good:

- For-loops with many executions: predict branch taken.
- If statements of the form

```
if(condition) {
   take care of the common case;
} else {
   handle unusual situation;
}
```

  – predict branch not taken (i.e. `then` clause will be executed).

Does Branch Prediction Work? (part 2)

Bad:

- Data dependent branches:

```
while((x < xtop) && (y < ytp)) { % merge-loop
    if(*x < *y) *(z++) = *(x++);
    else *(z++) = *(y++);
}
```

- If the data to be sorted is random
  - Any branch predictor will be wrong 50% of the time.
  - Even the mispredicted path can help reduce cache-misses.

Matrix Multiply on a Superscalar

2-cycles per loop iteration. One possible execution (simplified): on cycles $2i$ and $2i + 1$:

- One of the integer ALUs compute $\texttt{aptr}_i$ and $\texttt{bptr}_i$.
- The other integer ALU checks the branch prediction (branch taken) for iteration $i - 1$.
- The load store unit issues the loads for iteration $i - 1$.
  - Assume that the load-store unit can issue one load or store per cycle.
  - Assume cache accesses are pipelined and have a two cycle latency.
- The floating point multiplier issues the multiply for iteration $i - 2$.
  - I'll assume an R10K multiplier with a two-cycle latency.
- The floating point adder issues the add for iteration $i - 3$.

Matrix Multiplication, remarks

- The execution described above is 18 times faster than the original, microcoded machine.
- The superscalar is 2.5 times faster than the pipelined machine with loop unrolling.
- We can go even faster by:
  - Working on sub-blocks of the matrices that can be held in registers.
    - This gets around the load-bottlneck.
    - We now get one cycle per loop iteration.
  - Using a combined multiply-add operation.
    - If there are enough floating point registers to avoid a load-bottleneck,
    - We can now get two iterations per cycle!
    - However, this requires re-arranging floating point adds.
    - The compiler can't do this (floating point addition is not associative, due to round-off error).
    - High-performance scientific computing uses carefully, hand-tuned code to get high performance.
    - Automating such optimizations is a very cool area of active research.

. . . except when there's an exception

An exception causes the CPU to switch to executing a different sequence of instructions:

- system calls: trap into the operating system, switch from user to supervisor execution mode.
- page faults: the virtual address for a load or store doesn't have a corresponding physical memory location. Trap into the OS to bring the appropriate page in from disk and update the page tables.
- other exceptions caused by software: illegal addresses, overflow, illegal instruction. . . .
- interrupts $=$ exceptions caused by the hardware
  - I/O device interrupts (e.g. data ready from disk)
  - timer interrupts (e.g. the timer for the process scheduler)
  - hardware failures (e.g. uncorrectable memory read error)
  - some notebook computers have an accelerometer interrupt: "This laptop is falling, retract the disk heads before impact."

Precise Exceptions

- To restart a process after handling an exception, the OS needs to be able to reconstruct the process state. This also helps to identify the cause of the exception.
- An exception is precise if:
  - The exception is associated with a particular instruction.
  - All instructions (in program order) prior to the faulting instruction complete execution.
  - The faulting instruction and all subsequent ones appear to have never started.
- Programmers love precise exceptions, especially systems programmers ☺.
- Precise exceptions are fairly straightforward to implement on an "old-fashioned" CPU:
  - However, some complex instructions can cause challenges.
  - E.g. how do you handle a page-fault in the middle of a huge memory-block-copy instruction?

Precise Exceptions on a RISC Pipeline

- How to do it:
  - Let the faulting instruction continue to the write-back stage.
  - When it reaches write-back, cancel all instructions in previous stages.
    - This means making sure that the following instruction doesn't do a write in the `MEM` stage.
  - Handle the exception.
  - The process can resume by restarting at the faulting instruction.
- Exceptions and hazards:
  - Data hazards: must be handled by bypasses and stalls. Can't have the result from instruction `i` visible to instruction `i+1` only if instruction `i` faults.
  - Control hazards: record delay-slot info as part of the per-process data in the OS kernel.

Exceptions on a Superscalar

- Instructions graduate in program order $\Rightarrow$
  - Take the exception when the instruction would graduate.
- This ensures that all previous instructions have graduated.
- Abort all subsequent instructions.
  - This ensures that the current instruction and all subsequent ones appear to have never started.
  - Register restored by unwinding the register mappings to the point they were just before decoding and mapping the faulting instruction.
  - Memory has the correct values because stores are delayed until the store instruction graduates.

Superscalar Advantages

- “Free” parallelism
  - The programmer doesn’t have to explicitly find parallel code.
  - For example, the loop pipelining example from slide 10 happens automatically when executing the simple code for matrix multiply.
- Mitigates cache misses
  - Many applications are cache-miss limited.
  - A superscalar can execute beyond a cache miss to find the next miss.
  - The memory system can be pipelined and process multiple cache-misses concurrently:
    - This increases the bandwidth.
    - It does not lower the latency.
    - The superscalar processor can take advantage of the extra bandwidth.

Superscalar Scaling

Many structures on a super-scalar grow quadratically or worse with issue width:

- Register renaming: To rename W instructions at one time
  - Each of the instructions must compare its source register with the destination register of all previous instructions in the batch of W.
  - If they match, then the source is renamed to the match that destination;
  - Otherwise, the entry in the rename table is used.
  - This requires $\Omega(W^2)$ comparators.
- Register file size: The size of a register file is $O(NBP^2$, where N is the number of registers, B is the number of bits, and P is the number of read/write ports.
  - A register file has $O(NP)$ horizontal wires to select which registers to read or write;
  - $O(BP)$ vertical wires to connect the bits of those registers to the inputs and outputs of the register file.
  - P must grow in proportion to W, and N grows with W as well.
  - Register file area is $\Omega(W^2)$.

Furthermore, super-scalars are “communication maximizing” architectures:

- Every result gets broadcast to all issue queues that might need it.
- Moving bits around the CPU chip is the dominant source of power consumption,
- Power consumption is the limiting concern for CPU design.

Superscalar Summary

- Superscalars make sense in a world where wires are cheap and power is a secondary concern.
- That's not true any more.
- On the other hand, the benefit of instruction level parallelism (ILP) is big enough that general purpose computers seem to be heading modest issue width superscalar (peak of two or three instructions per cycle).
- Applications that are very power critical tend to use RISC style pipelined processors instead of super-scalars.
- Understanding super-scalars is important:
  - to understand the performance of programs running on real machines;
  - Because they provide a nice example of handling dependencies and using pipelining to improve throughput.

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.
  - footnotesize 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.

Preview

- Oct. 3: Shared Memory Multi-Processors
- Oct. 8: Message Passing Computers
- Oct. 10: Models of Parallel Compution.

Review

- What is instruction level parallelism?
- How does a pipelined architecture execute instruction in parallel?
- How does a superscalar machine handle dependencies?
  - RAW dependencies?
  - WAR dependencies?
  - WAW dependencies?
  - Control dependencies?
- How does a superscalar mitigate the performance impact of cache misses?
- Give one example of how the complexity of the hardware for a superscalar machine grows quadratically with the “issue width” of the machine.
- Do role models matter? What do you think? What has been your experience? What is the experience of other students you know at UBC?