# 5DV118
# Computer Organization and Architecture
# Umeå University
# Department of Computing Science

Stephen J. Hegner

## Topic 4: The Processor

### Part C: Other Designs for Instruction-Level Parallelism

These slides are mostly taken verbatim, or with minor changes, from those prepared by

Mary Jane Irwin (www.cse.psu.edu/~mji)

of The Pennsylvania State University

[Adapted from *Computer Organization and Design, 4th Edition*, Patterson & Hennessy, © 2008, MK]

## Key to the Slides

- The source of each slide is coded in the footer on the right side:
  - Irwin CSE331 = slide by Mary Jane Irwin from the course CSE331 (Computer Organization and Design) at Pennsylvania State University.
  - Irwin CSE431 = slide by Mary Jane Irwin from the course CSE431 (Computer Architecture) at Pennsylvania State University.
  - Hegner UU = slide by Stephen J. Hegner at Umeå University.

## Review: Pipeline Hazards

- Structural hazards
  - Design pipeline to eliminate structural hazards
- Data hazards – read before write
  - Use data forwarding inside the pipeline
  - For those cases that forwarding won't solve (e.g., load-use) include hazard hardware to insert stalls in the instruction stream
- Control hazards – `beq`, `bne,j,jr,jal`
  - Stall – hurts performance
  - Move decision point as early in the pipeline as possible – reduces number of stalls at the cost of additional hardware
  - Delay decision (requires compiler support) – not feasible for deeper pipes requiring more than one delay slot to be filled
  - Predict – with even more hardware, can reduce the impact of control hazard stalls even further if the branch prediction (BHT) is correct and if the branched-to instruction is cached (BTB)

## Extracting Yet *More* Performance

- Increase the depth of the pipeline to increase the clock rate – superpipelining
  - The more stages in the pipeline, the more forwarding/hazard hardware needed and the more pipeline latch overhead (i.e., the pipeline latch accounts for a larger and larger percentage of the clock cycle time)
- Fetch (and execute) more than one instructions at one time (expand every pipeline stage to accommodate multiple instructions) – multiple-issue
  - The instruction execution rate, CPI, will be less than 1, so instead we use IPC: instructions per clock cycle
    - E.g., a 6 GHz, four-way multiple-issue processor can execute at a peak rate of 24 billion instructions per second with a best case CPI of 0.25 or a best case IPC of 4
  - If the datapath has a five stage pipeline, how many instructions are active in the pipeline at any given time?

## Types of Parallelism

- Instruction-level parallelism (ILP) of a program – a measure of the average number of instructions in a program that a processor *might* be able to execute at the same time
  - Mostly determined by the number of true (data) dependencies and procedural (control) dependencies in relation to the number of other instructions
- Data-level parallelism (DLP)

```
DO   I = 1   TO   100
   A[I] = A[I] + 1
CONTINUE
```

- Machine parallelism of a processor – a measure of the ability of the processor to take advantage of the ILP of the program
  - Determined by the number of instructions that can be fetched and executed at the same time
- To achieve high performance, need *both* ILP and machine parallelism

## Multiple-Issue Processor Styles

- Static multiple-issue processors (aka VLIW)
  - Decisions on which instructions to execute simultaneously are being made statically (at compile time by the compiler)
  - E.g., Intel Itanium and Itanium 2 for the IA-64 ISA – EPIC (Explicit Parallel Instruction Computer)
    - 128-bit “bundles” containing three instructions, each 41-bits plus a 5-bit template field (which specifies which FU each instruction needs)
    - Five functional units (IntALU, Mmedia, Dmem, FPALU, Branch)
    - Extensive support for speculation and predication
- Dynamic multiple-issue processors (aka superscalar)
  - Decisions on which instructions to execute simultaneously (in the range of 2 to 8)  are being made dynamically (at run time by the hardware)
  - E.g., IBM Power series, Pentium 4, MIPS R10K, AMD Barcelona

## Multiple-Issue Datapath Responsibilities

- Must handle, with a combination of hardware and software fixes, the fundamental limitations of
  - How many instructions to issue in one clock cycle – issue slots
  - Storage (data) dependencies – aka data hazards
    - Limitation more severe in a SS/VLIW processor due to (usually) low ILP
  - Procedural dependencies – aka control hazards
    - Ditto, but even more severe
    - Use dynamic branch prediction to help resolve the ILP issue
  - Resource conflicts – aka structural hazards
    - A SS/VLIW processor has a much larger number of potential resource conflicts
    - Functional units may have to arbitrate for result buses and register-file write ports
    - Resource conflicts can be eliminated by duplicating the resource or by pipelining the resource

## Speculation

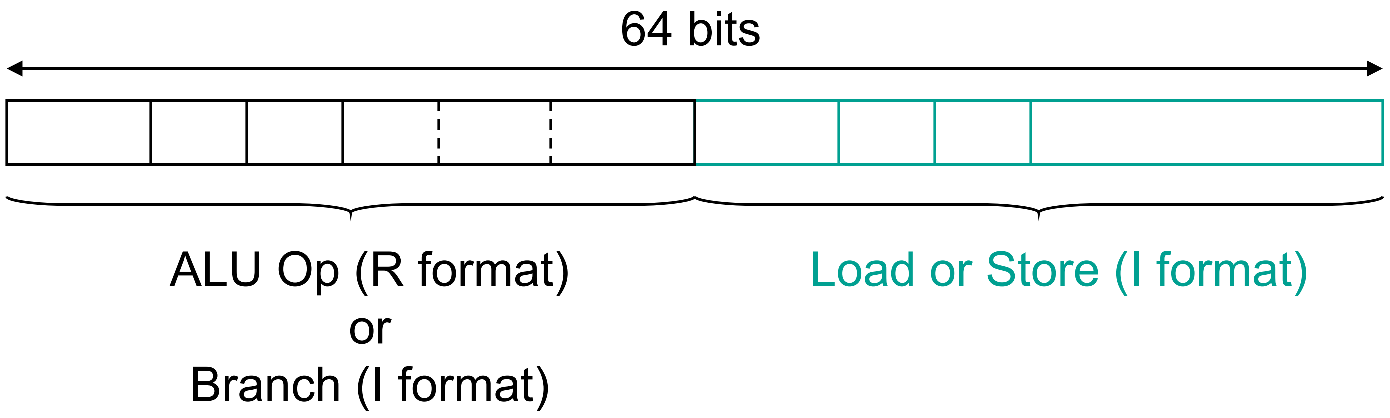
- Speculation is used to allow execution of future instr's that (may) depend on the speculated instruction
  - Speculate on the outcome of a conditional branch (branch prediction)
  - Speculate that a store (for which we don't yet know the address) that precedes a load does not refer to the same address, allowing the load to be scheduled before the store (load speculation)
- Must have (hardware and/or software) mechanisms for
  - Checking to see if the guess was correct
  - Recovering from the effects of the instructions that were executed speculatively if the guess was incorrect
- Ignore and/or buffer exceptions created by speculatively executed instructions until it is clear that they should really occur

## Static Multiple Issue Machines (VLIW)

- Static multiple-issue processors (aka VLIW) use the compiler (at compile-time) to statically decide which instructions to issue and execute simultaneously
  - Issue packet – the set of instructions that are bundled together and issued in one clock cycle – think of it as one large instruction with multiple operations
  - The mix of instructions in the packet (bundle) is usually restricted – a single "instruction" with several predefined fields
  - The compiler does static branch prediction and code scheduling to reduce (control) or eliminate (data) hazards
- VLIW's have
  - Multiple functional units
  - Multi-ported register files
  - Wide program bus

## An Example: A VLIW MIPS

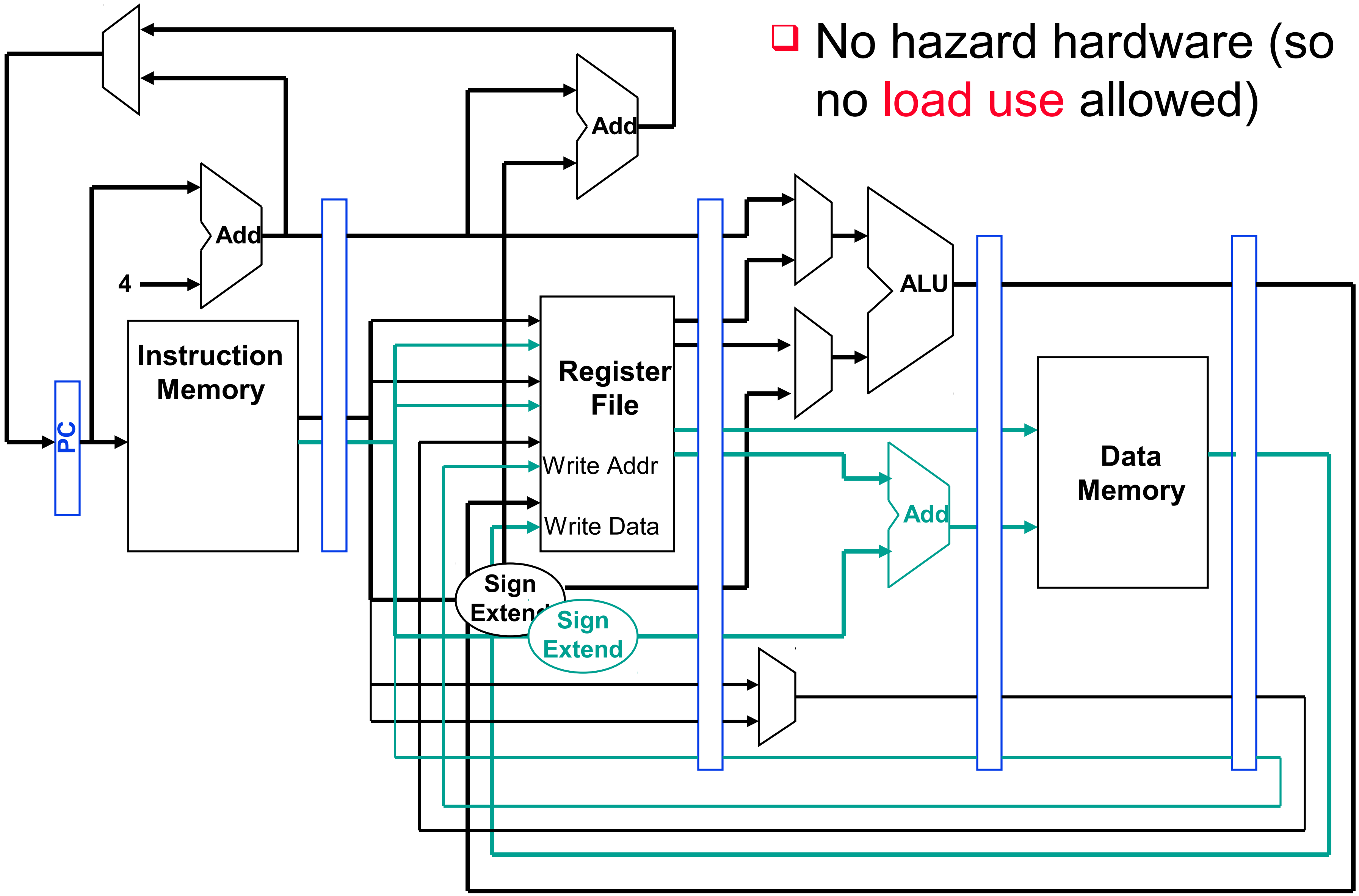
- Consider a 2-issue MIPS with a 2 instr bundle



- Instructions are always fetched, decoded, and issued in pairs
  - If one instr of the pair can not be used, it is replaced with a noop
- Need 4 read ports and 2 write ports and a separate memory address adder

# A MIPS VLIW (2-issue) Datapath

- No hazard hardware (so no load use allowed)



## Code Scheduling Example

- Consider the following loop code

```
lp:   lw    $t0,0($s1)    # $t0=array element
      addu  $t0,$t0,$s2   # add scalar in $s2
      sw    $t0,0($s1)    # store result
      addi  $s1,$s1,-4    # decrement pointer
      bne   $s1,$0,lp     # branch if $s1 != 0
```

- Must "schedule" the instructions to avoid pipeline stalls
  - Instructions in one bundle *must* be independent
  - Must separate load use instructions from their loads by one cycle
  - Notice that the first two instructions have a load use dependency, the next two and last two have data dependencies
  - Assume branches are perfectly predicted by the hardware

## The Scheduled Code (Not Unrolled)

| | ALU or branch | Data transfer | CC |
|---|---|---|---|
| lp: | | | 1 |
| | | | 2 |
| | | | 3 |
| | | | 4 |
| | | | 5 |

## The Scheduled Code (Not Unrolled)

| | ALU or branch | Data transfer | CC |
|---|---|---|---|
| `lp:` | | `lw   $t0,0($s1)` | 1 |
| | `addi  $s1,$s1,-4` | | 2 |
| | `addu  $t0,$t0,$s2` | | 3 |
| | `bne   $s1,$0,lp` | `sw   $t0,4($s1)` | 4 |
| | | | 5 |

- Four clock cycles to execute 5 instructions for a
  - CPI of 0.8 (versus the best case of 0.5)
  - IPC of 1.25 (versus the best case of 2.0)
  - noops don’t count towards performance !!

## Loop Unrolling

- Loop unrolling – multiple copies of the loop body are made and instructions from different iterations are scheduled together as a way to increase ILP

- Apply loop unrolling (4 times for our example) and then schedule the resulting code
  - Eliminate unnecessary loop overhead instructions
  - Schedule so as to avoid load use hazards

- During unrolling the compiler applies register renaming to eliminate all data dependencies that are not true data dependencies

## Unrolled Code Example

```
lp:   lw    $t0,0($s1)      # $t0=array element
      lw    $t1,-4($s1)     # $t1=array element
      lw    $t2,-8($s1)     # $t2=array element
      lw    $t3,-12($s1)    # $t3=array element
      addu  $t0,$t0,$s2     # add scalar in $s2
      addu  $t1,$t1,$s2     # add scalar in $s2
      addu  $t2,$t2,$s2     # add scalar in $s2
      addu  $t3,$t3,$s2     # add scalar in $s2
      sw    $t0,0($s1)      # store result
      sw    $t1,-4($s1)     # store result
      sw    $t2,-8($s1)     # store result
      sw    $t3,-12($s1)    # store result
      addi  $s1,$s1,-16     # decrement pointer
      bne   $s1,$0,lp       # branch if $s1 != 0
```

## The Scheduled Code (Unrolled)

| | ALU or branch | Data transfer | CC |
|---|---|---|---|
| `lp:` | `addi $s1,$s1,-16` | `lw $t0,0($s1)` | 1 |
| | | `lw $t1,12($s1)` | 2 |
| | `addu $t0,$t0,$s2` | `lw $t2,8($s1)` | 3 |
| | `addu $t1,$t1,$s2` | `lw $t3,4($s1)` | 4 |
| | `addu $t2,$t2,$s2` | `sw $t0,16($s1)` | 5 |
| | `addu $t3,$t3,$s2` | `sw $t1,12($s1)` | 6 |
| | | `sw $t2,8($s1)` | 7 |
| | `bne $s1,$0,lp` | `sw $t3,4($s1)` | 8 |

- Eight clock cycles to execute 14 instructions for a
  - CPI of 0.57 (versus the best case of 0.5)
  - IPC of 1.8 (versus the best case of 2.0)

## Predication

- Predication can be used to eliminate branches by making the execution of an instruction dependent on a "predicate", e.g.,

```
if (p) {statement 1} else {statement 2}
```

  would normally compile using two branches.  With predication it would compile as

```
 (p) statement 1
(~p) statement 2
```

- The use of `(condition)` indicates that the instruction is committed only if `condition` is true
- Predication can be used to speculate as well as to eliminate branches

## Compiler Support for VLIW Processors

- The compiler packs groups of independent instructions into the bundle
  - Done by code re-ordering (trace scheduling)
- The compiler uses loop unrolling to expose more ILP
- The compiler uses register renaming to solve name dependencies and ensures no load use hazards occur
- While superscalars use dynamic prediction, VLIW's primarily depend on the compiler for branch prediction
  - Loop unrolling reduces the number of conditional branches
  - Predication eliminates if-the-else branch structures by replacing them with predicated instructions
- The compiler predicts memory bank references to help minimize memory bank conflicts

## VLIW Advantages & Disadvantages

- Advantages
  - Simpler hardware (potentially less power hungry)
  - Potentially more scalable
    - Allow more instr's per VLIW bundle and add more FUs
- Disadvantages
  - Programmer/compiler complexity and longer compilation times
    - Deep pipelines and long latencies can be confusing (making peak performance elusive)
  - Lock step operation, i.e., on hazard all future issues stall until hazard is resolved (hence need for predication)
  - Object (binary) code incompatibility
  - Needs lots of program memory bandwidth
  - Code bloat
    - Noops are a waste of program memory space
    - Loop unrolling to expose more ILP uses more program memory space

## Dynamic Multiple Issue Machines (SS)

- Dynamic multiple-issue processors (aka SuperScalar) use hardware at run-time to dynamically decide which instructions to issue and execute simultaneously
- Instruction-fetch and issue – fetch instructions, decode them, and *issue* them to a FU to await execution
  - Defines the Instruction lookahead capability – fetch, decode and issue instructions beyond the current instruction
- Instruction-execution – as soon as the source operands and the FU are ready, the result can be calculated
  - Defines the processor lookahead capability – complete execution of issued instructions beyond the current instruction
- Instruction-commit – when it is safe to, write back results to the RegFile or D$ (i.e., change the machine state)

## In-Order vs Out-of-Order

- Instruction fetch and decode units are required to issue instructions in-order so that dependencies can be tracked
- The commit unit is required to write results to registers and memory in program fetch order so that
  - if exceptions occur the only registers updated will be those written by instructions before the one causing the exception
  - if branches are mispredicted, those instructions executed after the mispredicted branch don't change the machine state (i.e., we use the commit unit to correct incorrect speculation)
- Although the front end (fetch, decode, and issue) and back end (commit) of the pipeline run in-order, the FUs are free to initiate execution whenever the data they need is available – out-of-(program) order execution
  - Allowing out-of-order execution increases the amount of ILP

## Out-of-Order Execution

- With out-of-order execution, a later instruction may execute before a previous instruction so the hardware needs to resolve both read before write *and* write before write data hazards

```
lw    $t0,0($s1)
addu  $t0,$t1,$s2
. . .
sub   $t2, $t0, $s2
```

  - If the `lw` write to `$t0` occurs after the `addu` write, then the `sub` gets an incorrect value for `$t0`
  - The `addu` has an output dependency on the `lw` – write before write
    - The issuing of the `addu` might have to be stalled if its result could later be overwritten by an previous instruction that takes longer to complete

## Antidependencies

- Also have to deal with antidependencies – when a later instruction (that executes earlier) produces a data value that destroys a data value used as a source in an earlier instruction (that executes later)

```
R3 := R3 * R5
R4 := R3 + 1
R3 := R5 + 1
```

Antidependency
True data dependency
Output dependency

- The constraint is similar to that of true data dependencies, except *reversed*
  - Instead of the later instruction using a value (not yet) produced by an earlier instruction (read before write), the later instruction produces a value that destroys a value that the earlier instruction (has not yet) used (write before read)

## Dependencies Review

- Each of the three data dependencies
  - True data dependencies (read before write)
  - Antidependencies (write before read)
  - Output dependencies (write before write)

  } storage conflicts

  manifests itself through the use of registers (or other storage locations)

- True dependencies represent the flow of data and information through a program

- Anti- and output dependencies arise because the limited number of registers mean that programmers reuse registers for different computations leading to storage conflicts

## Storage Conflicts and Register Renaming

- Storage conflicts can be reduced (or eliminated) by increasing or duplicating the troublesome resource
  - Provide additional registers that are used to reestablish the correspondence between registers and values
    - Allocated dynamically by the hardware in SS processors

- Register renaming – the processor renames the original register identifier in the instruction to a new register (one not in the visible register set)

```
R3 := R3 * R5
R4 := R3 + 1
R3 := R5 + 1
```

$\Longrightarrow$

```
R3b := R3a * R5a
R4a := R3b + 1
R3c := R5a + 1
```

  - The hardware that does renaming assigns a “replacement” register from a pool of free registers and releases it back to the pool when its value is superseded and there are no outstanding references to it

## Summary: Extracting More Performance

- To achieve high performance, need both machine parallelism and instruction level parallelism (ILP) by
  - Superpipelining
  - Static multiple-issue (VLIW)
  - Dynamic multiple-issue (superscalar)

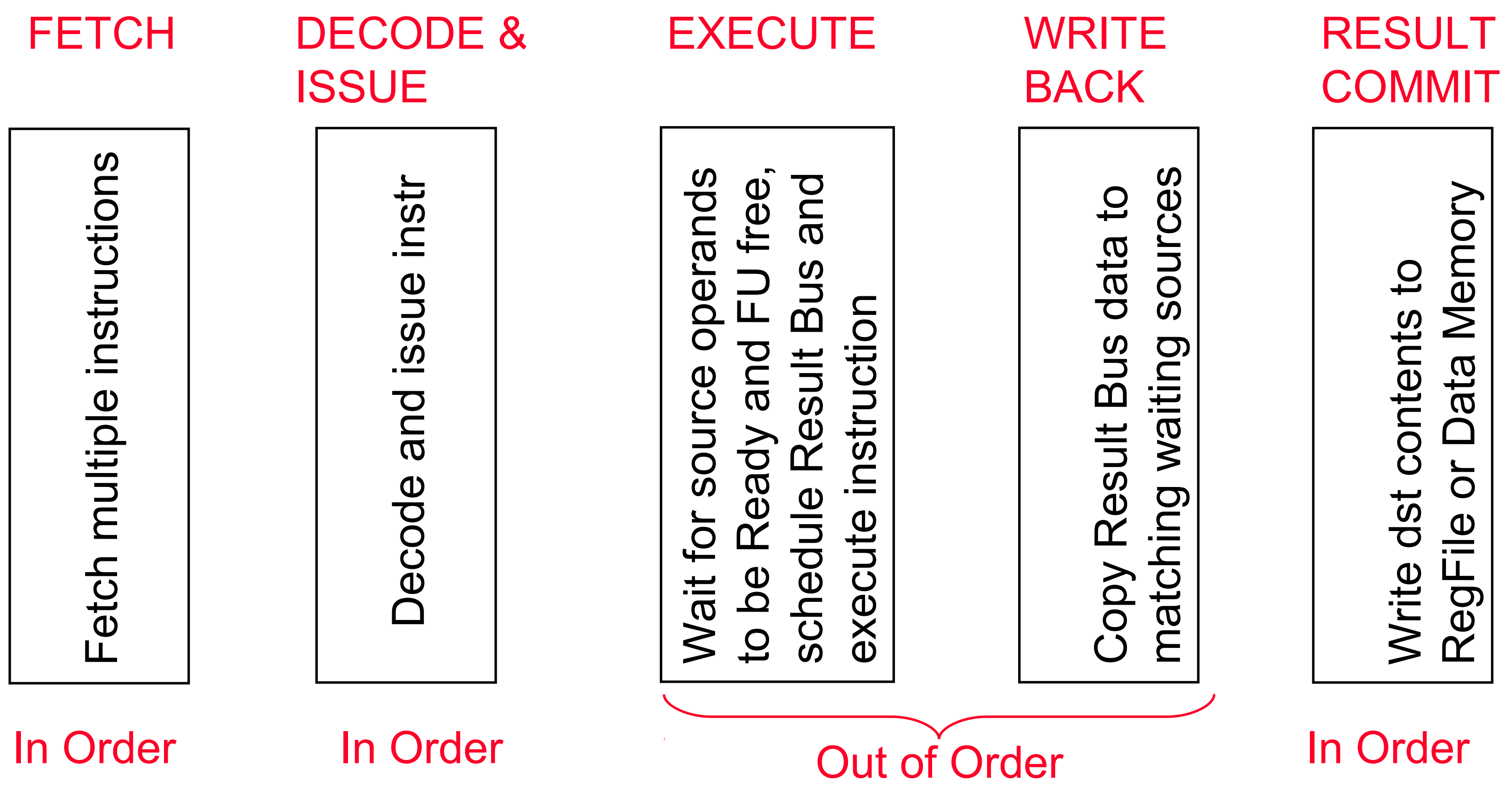
- A processor's instruction issue and execution policies impact the available ILP
  - In-order fetch, issue, and commit and out-of-order execution
    - Pipelining creates **true** dependencies (read before write)
    - Out-of-order execution creates **antidependencies** (write before read)
    - Out-of-order execution creates **output dependencies** (write before write)
    - In-order commit allows speculation (to increase ILP) and is required to implement precise interrupts

- Register renaming can solve these storage dependencies

## SimpleScalar Structure

- `sim-outorder`: supports out-of-order execution (with in-order commit) with a Register Update Unit (RUU)
  - Uses a RUU for register renaming and to hold the results of pending instructions. The RUU (aka reorder buffer (ROB)) retires (i.e., commits) completed instructions in program order to the RegFile
  - Uses a LSQ for store instructions not ready to commit and load instructions waiting for access to the D$
  - Loads are satisfied by either the memory or by an earlier store value residing in the LSQ if their addresses match
    - Loads are issued to the memory system only when addresses of *all* previous loads and stores are known

# SS Pipeline Stage Functions


FETCH
DECODE & ISSUE
EXECUTE
WRITE BACK
RESULT COMMIT
Fetch multiple instructions
Decode and issue instr
Wait for source operands to be Ready and FU free, schedule Result Bus and execute instruction
Copy Result Bus data to matching waiting sources
Write dst contents to RegFile or Data Memory
In Order
In Order
Out of Order
In Order

```
ruu_fetch()                           ruu_issue()                  ruu_commit()
            ruu_dispatch()            lsq_refresh()
                                                   ruu_writeback()
```

## Simulated SimpleScalar Pipeline

- `ruu_fetch()`: fetches instr's from one I$ line, puts them in the fetch queue, probes the cache line predictor to determine the next I$ line to access in the next cycle
  - fetch:ifqsize<size>: fetch width (default is 4)
  - fetch:speed<ratio>: ratio of the front end speed to the execution core (<ratio> times as many instructions fetched as decoded per cycle)
  - fetch:mplat<cycles>: branch misprediction latency (default is 3)
- `ruu_dispatch()`: decodes instr's in the fetch queue, puts them in the dispatch (scheduler) queue, enters and links instr's into the RUU and the LSQ, splits memory access instructions into two separate instr's (one to compute the effective addr and one to access the memory), notes branch mispredictions
  - decode:width<insts>: decode width (default is 4)

## SimpleScalar Pipeline, con't

- `ruu_issue()` and `lsq_refresh()`: locates and marks the instr's ready to be executed by tracking register and memory dependencies, ready loads are issued to D$ unless there are earlier stores in LSQ with unresolved addr's, forwards store values with matching addr to ready loads
  - issue:width<insts>: maximum issue width (default is 4)
  - ruu:size<insts>: RUU capacity in instr's (default is 16, min is 2)
  - lsq:size<insts>: LSQ capacity in instr's (default is 8, min is 2)

  and handles instr's execution – collects all the ready instr's from the scheduler queue (up to the issue width), check on FU availability, checks on access port availability, schedules writeback events based on FU latency (hardcoded in `fu_config[]`)
  - res:ialu | imult | memport | fpalu | fpmult<num>: number of FU's (default is 4 | 1 | 2 | 4 | 1)

## SimpleScalar Pipeline, con't

- `ruu_writeback()`: determines completed instr's, does data forwarding to dependent waiting instr's, detects branch misprediction and on misprediction rolls the machine state back to the checkpoint and discards erroneously issued instructions

- `ruu_commit()`: in-order commits results for instr's (values copied from RUU to RegFile or LSQ to D$), RUU/LSQ entries for committed instr's freed; keeps retiring instructions at the head of RUU that are ready to commit until the head instr is one that is not ready

## CISC vs RISC vs SS vs VLIW

| | CISC | RISC | Superscalar | VLIW |
|---|---|---|---|---|
| **Instr size** | variable size | fixed size | fixed size | fixed size (but large) |
| **Instr format** | variable format | fixed format | fixed format | fixed format |
| **Registers** | few, some special<br>Limited # of ports | Many GP<br>Limited # of ports | GP and rename (RUU)<br>Many ports | many, many GP<br>Many ports |
| **Memory reference** | embedded in many instr's | load/store | load/store | load/store |
| **Key Issues** | decode complexity | data forwarding, hazards | hardware dependency resolution | (compiler) code scheduling |

## Evolution of Pipelined, SS Processors

| | Year | Clock Rate | # Pipe Stages | Issue Width | OOO? | Cores /Chip | Power |
|---|---|---|---|---|---|---|---|
| Intel 486 | 1989 | 25 MHz | 5 | 1 | No | 1 | 5 W |
| Intel Pentium | 1993 | 66 MHz | 5 | 2 | No | 1 | 10 W |
| Intel Pentium Pro | 1997 | 200 MHz | 10 | 3 | Yes | 1 | 29 W |
| Intel Pentium 4 Willamette | 2001 | 2000 MHz | 22 | 3 | Yes | 1 | 75 W |
| Intel Pentium 4 Prescott | 2004 | 3600 MHz | 31 | 3 | Yes | 1 | 103 W |
| Intel Core | 2006 | 2930 MHz | 14 | 4 | Yes | 2 | 75 W |
| Sun USPARC III | 2003 | 1950 MHz | 14 | 4 | No | 1 | 90 W |
| Sun T1 (Niagara) | 2005 | 1200 MHz | 6 | 1 | No | 8 | 70 W |