VLIW processors

- Superscalar machines use hardware to reorder instructions and keep functional units busy
- In VLIW (Very Long Instruction Word) machines, all of this burden falls upon the compiler
  - Each VLIW “instruction” is composed of multiple independent instructions, each of which execute on different function units
    - Functional units might include integer ALUs, FP ALUs, memory units, and a branch unit
    - The instruction must allocate 16 or more bits to each unit to describe the operation that the unit will run on each cycle
  - To keep the functional units busy, parallelism is uncovered by the compiler by unrolling loops and scheduling code across basic blocks
- A VLIW CPU can also help by providing forwarding

Sample VLIW processor

- VLIW machine that can issue **two** memory references, **two** FP operations, and **one** integer/branch operation per clock cycle
- Loop unrolled 7 times
  - Ignoring branch delay, loop achieves 2.5 operations per clock
  - Total time is 9 cycles for 7 iterations

Memory 1	Memory 2	FP 1	FP 2	Integer/branch
`LD F0,0(R1)`	`LD F6,-8(R1)`			
`LD F10,-16(R1)`	`LD F14,-24(R1)`			
`LD F18,-32(R1)`	`LD F22,-40(R1)`	`ADDD F4,F0,F2`	`ADDD F8,F6,F2`	
`LD F26,-48(R1)`		`ADDD F12,F10,F2`	`ADDD F16,F14,F2`	
		`ADDD F20,F18,F2`	`ADDD F24,F22,F2`	
`SD 0(R1),F4`	`SD -8(R1),F8`	`ADDD F28,F26,F2`		
`SD -16(R1),F12`	`SD -24(R1),F16`			
`SD -32(R1),F20`	`SD -40(R1),F24`			`SUBI R1,R1,#56`
`SD 8(R1),F28`				`BNEZ R1,Loop`

Limits in multiple-issue processors

- Why stop at 5 instructions/clock? Why not 50?
- Limits on available ILP in programs
  - There are usually not enough operations to fill all of the available slots
  - It might seem that 5 independent instructions are sufficient in the example; however, the memory, branch and FP units will likely be pipelined and have a multicycle latency
    - Assume a latency of 6 clocks for the FP units, and that two FP pipelined units are available
    - This requires that there are 12 FP instructions that are independent of the most recently issued FP instruction!
  - If a branch requires just a one cycle latency, it results in a 5 instruction latency in the example CPU machine

Limits in multiple-issue processors

- Hardware complexity
  - Additional functional units: duplicate integer and FP units for multiple-issue
    - Their cost scales linearly
  - Added bandwidth to registers
    - More register file ports are required to sustain the multiple issue
    - A single integer pipeline requires 3 ports to a register file
    - Adding another pipeline requires 3 more ports
  - Added memory ports: necessary for multiple memory units
    - Much more expensive than register ports
  - Scheduling hardware
    - Relatively simple for VLIW
    - Can be very complex for superscalar architectures

Limits in multiple-issue processors

- Superscalar CPUs have complex instruction issue logic
- VLIW CPUs have other problems
  - Technical problems
    - Increase in code size from open slots (wasted bits for unused functional units) increases memory bandwidth requirements
    - A stall (i.e., cache miss) in any functional unit causes the entire processor to stall because of the lock step operation of VLIW
  - Logistical problems
    - Binary compatibility is a problem because adding functional units or changing latencies requires major code changes
- Complexity and access time penalties of a multiported memory hierarchy are probably the most serious hardware limitations of superscalar and VLIW implementations

Even more parallelism

- Previously discussed methods that the compiler can use to discover ILP
  - Works as long as branch behavior is relatively predictable
- Better: increase levels of ILP in programs
  - Conditional execution: instructions that are “executed” only when a certain condition holds
  - Speculative execution
    - Execute instructions that might be needed later
    - Example: execute both forks of a branch

Conditional instructions

- A conditional instruction refers to a condition which is evaluated as part of the instruction execution
  - Don’t use a branch to skip a single instruction
  - Instr always executes but only writes the result if the condition is met
- Eliminating the branch gives two benefits
  - The branch is not executed, reducing the instruction count by 1
  - The branch delay is avoided
- Conditional execution changes a control dependence into a data dependence
  - In an integer pipeline, **data** dependencies rarely cause stalls while **control** hazards do cause stalls

```
                         BNEZ R1,L
if (A==0)                MOV  R2,R3                CMOVZ R2,R3,R1
  S = T;      ──────▶                   ──────▶
                       L:
```

Benefits of conditional instructions

- Conditional instructions help a lot with superscalar machines because such machines suffer even more from branch stalls
  - Conditional instructions can be scheduled as normal instructions
  - Branches often cannot be scheduled this way because they may cause a change in the instruction stream

  ⇒ More slots in a superscalar machine can be filled
- Conditional instructions are of even greater benefit on a VLIW machine for similar reasons

Conditional instructions & exceptions

- Conditional instructions must not introduce an exception if its condition isn't satisfied
  - The instruction must have NO effect if the condition is not satisfied
  - In example below, if R10 contains *zero*, it's likely that the LW instruction will cause a protection violation if allowed to execute
- Solution:
  - In DLX, memory accesses are not started until MEM
  - It's easy to evaluate the condition (i.e. during EX) and prevent the memory access from happening in this case

```
  BEQZ R10,L
  LW   R8,20(R10)  ———————>  LWC R8,20(R10),R10
L:
```

Limits to conditional instructions

- Executing conditional instructions takes time
  - A conditional instruction always requires time, even if the instruction is annulled
  - Moving an instruction across a branch is essentially speculating on the outcome of the branch
  - May slow down a program if an instruction is executed but turned into a no-op, since another instruction may have executed during that slot
  - Conditional instructions are always a win when the cycle that they occupy would have been idle anyway
- Sequence length can affect performance
  - Trading a branch and move for a conditional move is usually a win
  - Longer sequences may not be

Limits to conditional instructions

- The condition must be evaluated early
  - The condition must be known before the processor’s state is changed, and the earlier the better
- Conditional instructions are difficult for multiple conditions
  - These instructions work well for avoiding single branches
  - The task is more difficult for two or more branch options: it requires additional instructions to logically combine the multiple conditions
- Conditional instructions may impose a speed penalty
  - The cycle time for the entire CPU might be increased
  - A conditional instruction might take more clock cycles to execute than a non-conditional instruction

Compiler-directed speculative execution

- Conditional instructions eliminate control dependencies for small if-then blocks
- Moving larger blocks of code across (before) branches can yield larger performance gains
- Doing so creates problems in two areas
  - Registers that should not be modified (because of the branch) are modified anyway
  - Exceptions that should not occur may actually happen (as with conditional instructions)
- Resumable exceptions such as page faults aren’t a big problem
  - May cause performance to suffer somewhat
  - Programs don’t terminate incorrectly

Implementing speculation: ignore exceptions

- **Three** schemes for supporting speculation without introducing erroneous exception behavior have been investigated
- First scheme: ignore exceptions
  - The simplest method for speculation is for the CPU and OS to ignore non-resumable exceptions for speculative instructions
  - Rather than terminate the program, they return an undefined value for the instruction causing the exception
    - If the exception generating instruction was not speculative, the program is in error but it is allowed to continue!
      - ⇒However, it'll probably generate incorrect results
    - If the exception generating instruction was speculative, the speculative result won't be used and the program will run properly
  - Either way, a correct program is not terminated improperly

Ignore exceptions: example

```
If (A==0)
  A = B;
else
  A += 4;
```

```
    LW   R1,0(R3) ; load A
    BNEZ R1,L1    ; test A
    LW   R1,0(R2) ; if clause
    J    L2       ; skip else
L1: ADDI R1,R1,#4 ; else clause
L2: SW   0(R3),R1 ; store A
```

Two possibilities assuming **then** almost always executed

```
    LW   R1,0(R3)
    LW   R14,0(R2)
    BEQZ R1,L3
    ADDI R14,R1,#4
L3: SW   0(R3),R1
```

Speculative load B

Non-speculative store

```
LW   R1,0(R3)
LW   R14,0(R2)
ADDI R16,R1,#4
MVC  R14,R16,R1
SW   0(R3),R1
```

- Why use R14?
- Where is the value of A at the end of this sequence?

- Branch replaced by conditional move
- Under what circumstances is this more expensive?

Speculative execution: poison bits

- Each register has a "poison bit" attached to it
  - If a speculative instruction causes an exception, the exception is handled by setting the poison bit of its destination register
  - If another speculative instruction uses a poisoned register as a source operand, its destination register poison bit is also set
  - If a non-speculative instruction uses a poisoned register, an exception is generated
    - It may, however, write to a poisoned register
    - If this occurs, the poison bit is cleared
- This method generates exceptions for incorrect programs (at about the right place)

  ⇒ The OS must be able to save, restore, and reset the poison bits, which requires special instructions

Speculative execution: boosting

- Previous schemes introduced register copies
- This approach (called boosting) provides ***renaming*** and ***buffering*** in hardware, similar to Tomasulo's approach
  - A boosted instruction is executed speculatively based on a branch
  - Its results are forwarded to and used by other boosted instructions
  - When the branch is reached, the results are committed to the register file if the prediction is correct
- Therefore, instructions that are control dependent on a branch can be executed **before the branch**

Speculative execution with renaming

```
If (A==0)
  A = B;
else
  A += 4;
```

```
     LW   R1,0(R3) ; load A
     BNEZ R1,L1    ; test A
     LW   R1,0(R2) ; if clause
     J    L2       ; skip else
L1:  ADDI R1,R1,#4 ; else clause
L2:  SW   0(R3),R1 ; store A
```

+ indicates the instruction is boosted across the next branch and predicts the branch taken

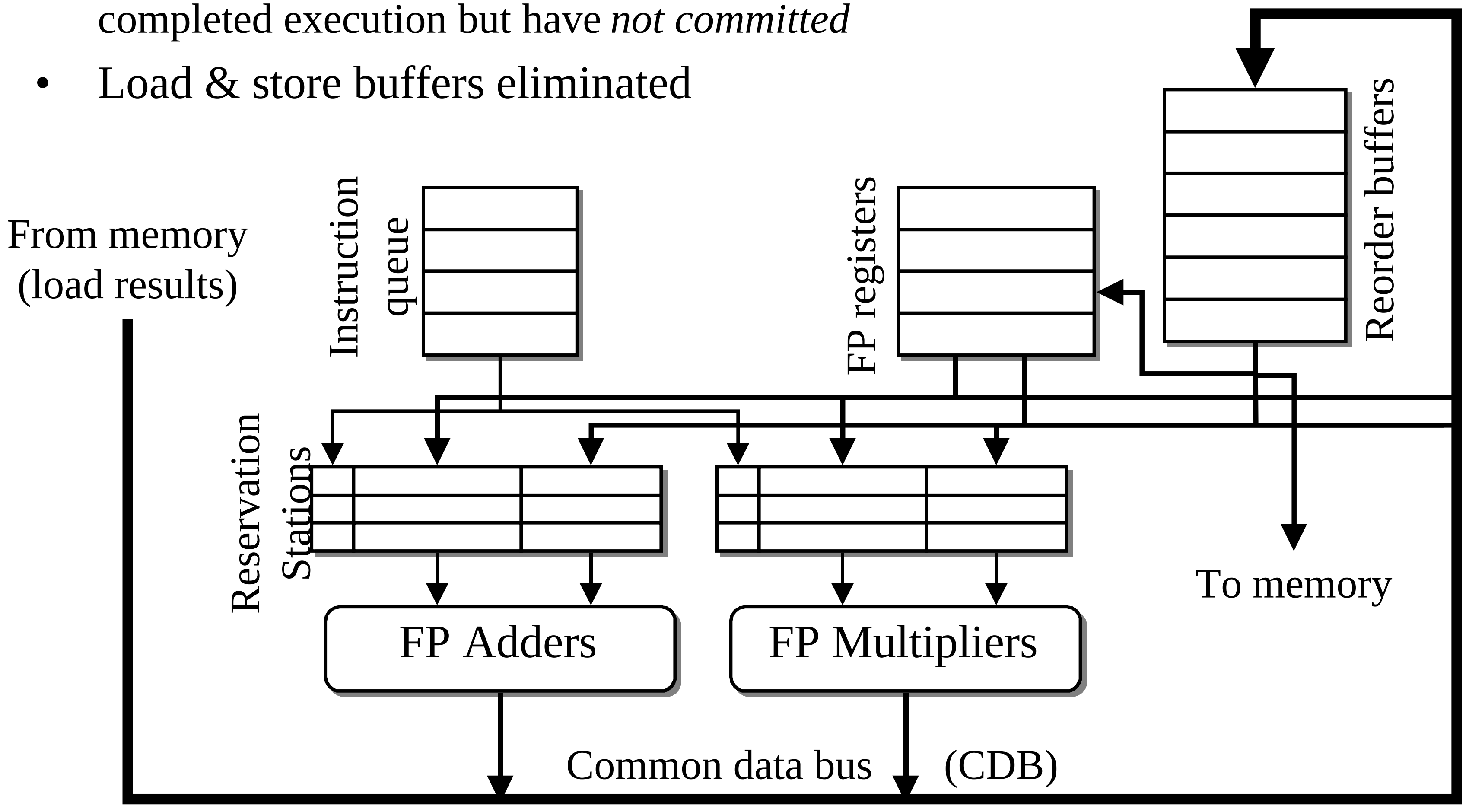
```
     LW   R1,0(R3)
     LW+  R1,0(R2) ; boosted load of B
     BEQZ R1, L3
     ADDI R1,R1,#4
L3:  SW   0(R3),R1 ; non-speculative store
```

Hardware-based speculation

- Combine speculative execution and dynamic scheduling based on Tomasulo's approach
  - Focus on floating-point operations
  - Similar structures can handle integer operations
- Change Tomasulo's approach to support speculation
  - Separate the process of completing execution and the bypassing of results among instructions from instruction commit (register file or memory update)
  - This allows other (speculative) instructions to execute, but no results are committed until we know the instruction is no longer speculative
- Allow instructions to execute out of order but force them to commit in order
  - Helps handle exceptions properly

Hardware-based speculation: design

- A set of hardware buffers (***reorder buffers***) hold the results of instructions that have completed execution but have *not committed*
- Load & store buffers eliminated



Hardware-based speculation: stages

- The reorder buffer provides additional virtual registers and is a source of operands for instructions
- An additional step is added to Tomasulo's algorithm
  - Issue
    - Get a floating-point instruction
    - Issue it if there is a reservation station open and an empty slot in the reorder buffer
    - Send the number of the reorder buffer assigned for the result to the reservation station so it can be used to tag the result
  - Execute
  - Monitor the CDB while waiting for source registers to be ready
  - When both operands are available, perform the operation

Hardware-based speculation: stages

- More steps in to Tomasulo’s algorithm
- Write result
  - Write the result on the CDB with the reorder buffer tag
  - Result is stored into the reorder buffer as well as into any reservation stations waiting for the result
  - Reorder buffer can also serve as a source register for operands similar to the register file
- Commit
  - When the instruction reaches the head of the reorder buffer and its result is present in the buffer, update the register or write memory
  - When an incorrectly predicted branch arrives, flush the reorder buffer and restart execution at the correct successor of the branch
  - If the branch was correctly predicted, do nothing

Hardware-based speculation: advantages

- This scheme has several advantages over dynamic scheduling alone
- Instructions can “finish” out of order as long as they are not committed
  - ⇒ The CPU can keep *precise interrupts* even while executing out of order since changes are committed in order.
- The CPU to *speculatively* execute instructions past a branch before the branch is executed
  - ⇒ Instructions are canceled if the branch is mispredicted
- Handle exceptions just before the instruction is ready to commit
  - – All previous instructions and no later instructions have committed
  - – The CPU can do a precise exception even with out-of-order execution

Speculation & multiple-issue CPUs

- The techniques that work in single-issue CPUs work in multiple-issue CPUs as well
  - Speculate on both integer and floating point instructions
  - More complex design
    - More hazards to check for
    - CDB (maybe more than one!) gets crowded...
- Speculation may be more useful in such processors
  - Longer branch delays and operation latencies
  - More empty execution slots that speculation can fill (potentially usefully)

Lower CPI isn’t always faster

- If the lower CPI comes at the expense of a longer clock cycle, it may slow the processor down
  - Almost invariably true since lowering CPI using hardware means implementing more sophisticated techniques which increase clock cycle time
- This inclination arises because
  - Simulation tools to evaluate the impact of enhancements that affect CPI are more readily available than tools to evaluate the impact on clock cycle time
  - Accurate analysis on the impact of clock rate is not usually possible until the design is well underway

Improve the whole CPU, not just part

- As with uniprocessors, improving one aspect of a CPU does not help unless it was the bottleneck from the beginning
  - Improving FP latency for a multiple-issue CPU does not help much unless something is done about branching
  - Making branches faster doesn't help if the CPU stalls a lot waiting for integer hazards
- Speculative execution is great but is of limited benefit unless there are additional registers to use
  - Under compiler control (larger register set)
  - "Virtual" registers used by dynamic scheduler