Lecture 12
Processor Microarchitecture (Part 3)

Xuan 'Silvia' Zhang
Washington University in St. Louis

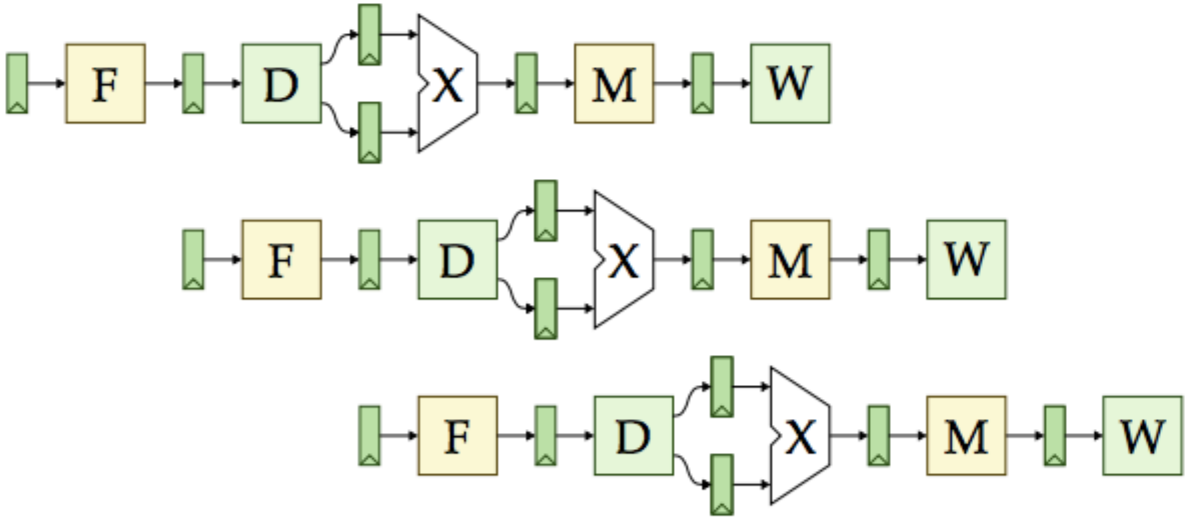
http://classes.engineering.wustl.edu/ese566/

Pipeline Hazards: RAW Data Hazards

RAW data hazards occur when one instruction depends on a data value produced by a preceding instruction still in the pipeline. We use architectural dependency arrows to illustrate RAW dependencies in assembly code sequences.

```
addiu r1, r2, 1

addiu r3, r1, 1

addiu r4, r3, 1
```

```
addiu r1, r2, 1

addiu r3, r1, 1

addiu r4, r3, 1
```



Approaches to Resolving Data Hazards

- Expose in instruction set architecture
  - expose data hazards in ISA forcing compiler to explicitly avoid scheduling instructions that would create hazards (i.e. software scheduling for correctness)
- Hardware Scheduling
  - hardware dynamically schedules instruction to avoid RAW hazards, potentially allowing instructions to execute out of order
- Hardware Stalling
  - hardware includes control logic that freezes later instructions until earlier instruction has finished producing data value; software scheduling can still be used to avoid stalling (i.e. software scheduling for performance)

Approaches to Resolving Data Hazards

- Hardware bypassing/forwarding
  - hardware allows values to be sent from an earlier instruction to a later instruction before the earlier instruction has left the pipeline
- Hardware speculation
  - hardware guesses that there is no hazard and allows later instructions to potentially read invalid data; detects when there is a problem, squashes and then re-executes instructions that operated on invalid data

Expose in ISA

Insert nops to delay read of earlier write. These nops count as real instructions increasing instructions per program.

```
addiu r1, r2, 1
nop
nop
nop
addiu r3, r1, 1
nop
nop
nop
addiu r4, r3, 1
```

Insert independent instructions to delay read of earlier write, and only use nops if there is not enough useful work

```
addiu r1, r2, 1
addiu r6, r7, 1
addiu r8, r9, 1
nop
addiu r3, r1, 1
nop
nop
nop
addiu r4, r3, 1
```

Hardware Stalling

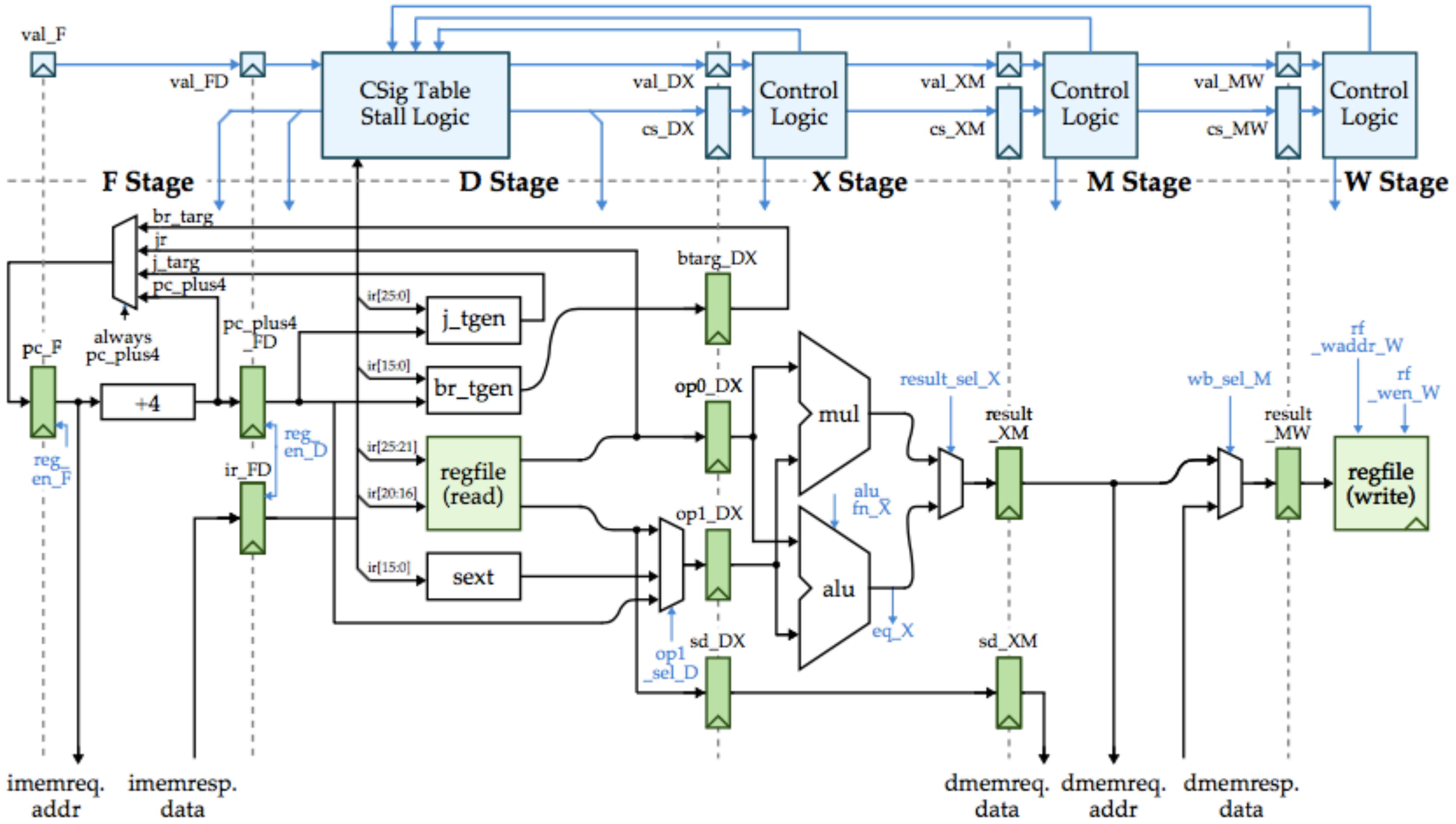
Hardware includes control logic that freezes later instructions (in front of pipeline) until earlier instruction (in back of pipeline) has finished producing data value.

Pipeline diagram showing hardware stalling for RAW data hazards

`addiu r1, r2, 1`														
`addiu r3, r1, 1`														
`addiu r4, r3, 1`														

Note: Software scheduling is not required for correctness, but can improve performance! Programmer or compiler schedules independent instructions to reduce the number of cycles spent stalling.

Datapath to Support Hardware Stalling



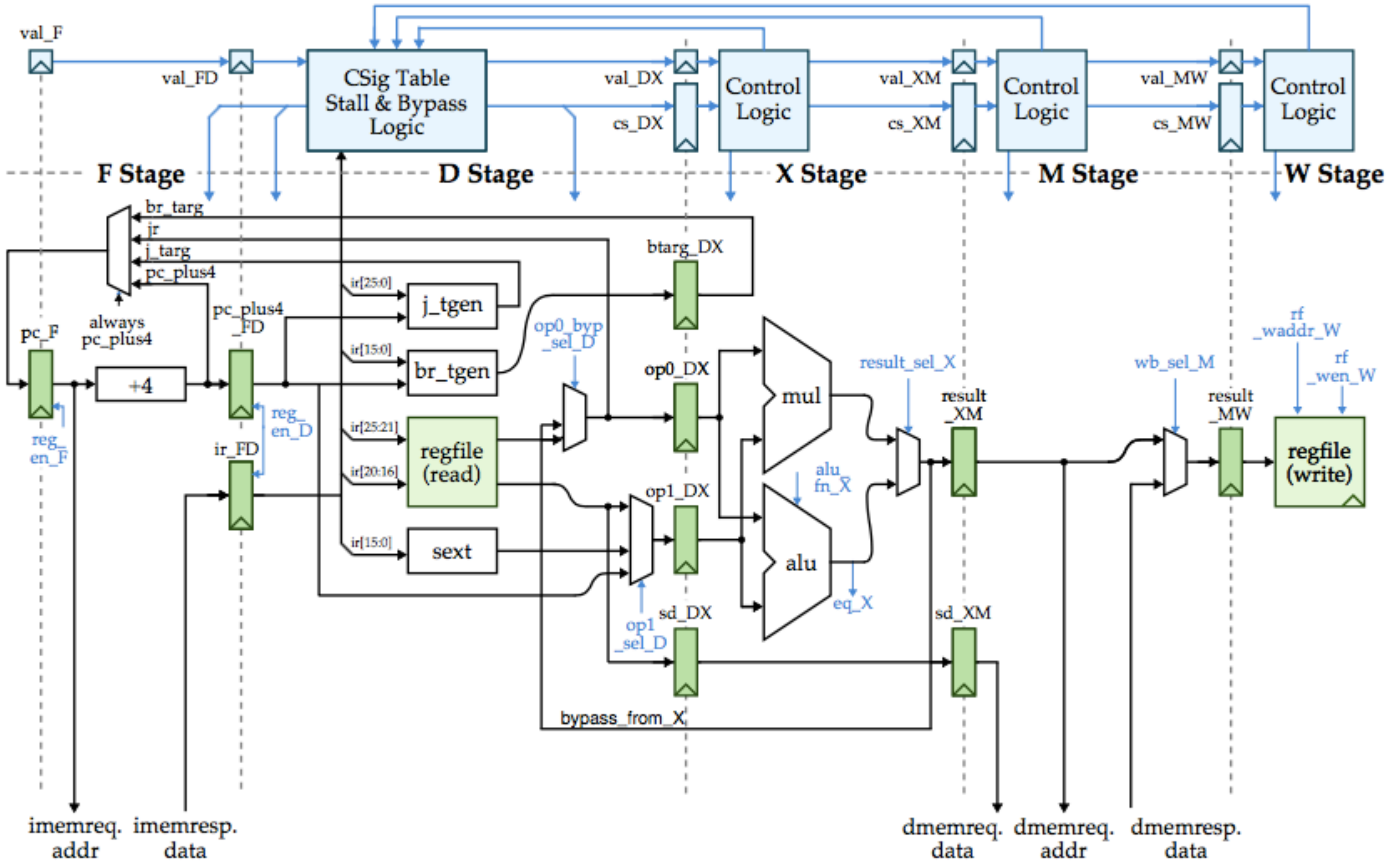
Hardware Bypassing/Forwarding

Hardware allows values to be sent from an earlier instruction (in back of pipeline) to a later instruction (in front of pipeline) before the earlier instruction has left the pipeline. Sometimes called “forwarding”.

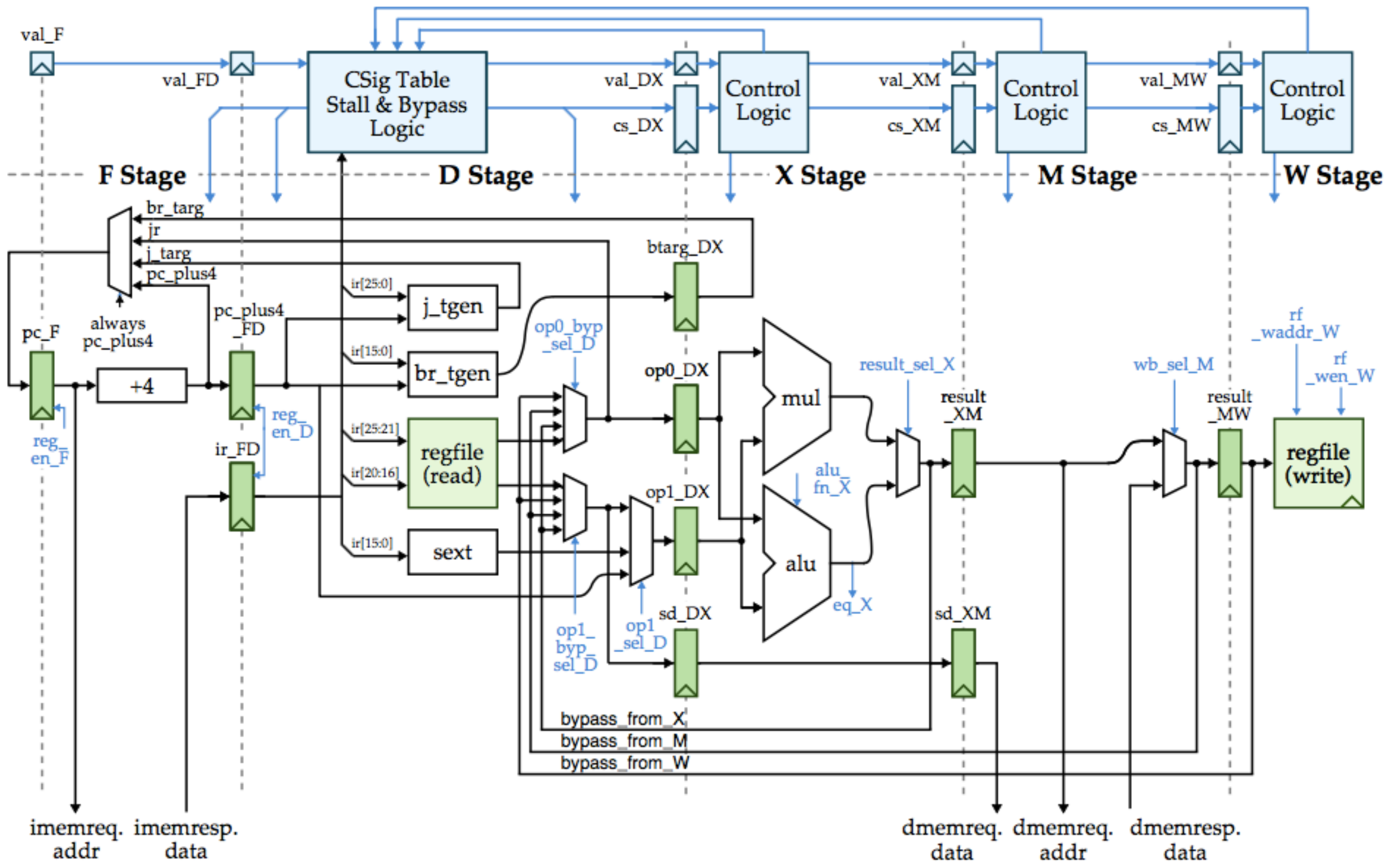
Pipeline diagram showing hardware bypassing for RAW data hazards

`addiu r1, r2, 1`														
`addiu r3, r1, 1`														
`addiu r4, r3, 1`														

Add Single Bypass Path



Add All Bypass Paths



RAW Data Hazards Through Memory

So far we have only studied RAW data hazards through registers, but we must also carefully consider RAW data hazards through memory.

```
sw r1, 0(r2)
lw r3, 0(r4) # RAW dependency occurs if R[r2] == R[r4]
```

sw r1, 0(r2)														
lw r3, 0(r4)														

Pipeline Hazards: Control Hazards

Control hazards occur when whether or not an instruction should be executed depends on a control decision made by an earlier instruction We use architectural dependency arrows to illustrate control dependencies in assembly code sequences.

Static Instr Sequence

```
      addiu r1, r0, 1
      j     foo
      opA
      opB
foo:  addiu r2, r3, 1
      bne   r0, r1, bar
      opC
      opD
      opE
bar:  addiu r4, r5, 1
```

Dynamic Instr Sequence

```
addiu r1, r0, 1
j     foo
addiu r2, r3, 1
bne   r0, r1, bar
addiu r4, r5, 1
```

Pipeline Hazards: Control Hazards

The jump resolution latency and branch resolution latency are the number of cycles we need to delay the fetch of the next instruction in order to avoid any kind of control hazard. Jump resolution latency is two cycles, and branch resolution latency is three cycles.

`addiu r1, r0, 1`														
`j foo`														
`addiu r2, r3, 1`														
`bne r0, r1, bar`														
`addiu r4, r5, 1`														

Approaches to Resolving Control Hazards

- Expose in ISA
  - expose control hazards in ISA forcing compiler to explicitly avoid scheduling instructions that would create hazards (i.e., software scheduling for correctness)
- Software predication
  - programmer or compiler converts control flow into data flow by using instructions that conditionally execute based on a data value

Approaches to Resolving Control Hazards

- Hardware speculation
  - hardware guesses which way the control flow will go and potentially fetches incorrect instruction; detects when there is a problem and re-executes instructions along the correct control flow
- Software hints
  - programmer or compiler provides hints about whether a conditional branch will be taken or not taken, and hardware can use these hints for more efficient hardware speculation

Expose in ISA

Expose branch delay slots as part of the instruction set. Branch delay slots are instructions that follow a jump or branch and are *always* executed regardless of whether a jump or branch is taken or not taken. Compiler tries to insert useful instructions, otherwise inserts nops.

```
     addiu r1, r0, 1
     j     foo
     nop
     opA
     opB
foo: addiu r2, r3, 1
     bne   r0, r1, bar
     nop
     nop
     opC
     opD
     opE
bar: addiu r4, r5, 1
```

Assume we modify the PARCv1 instruction set to specify that J, JAL, and JR instructions have a single-instruction branch delay slot (i.e., one instruction after a J, JAL, and JR is always executed) and the BNE instruction has a two-instruction branch delay slot (i.e., two instructions after a BNE are always executed).

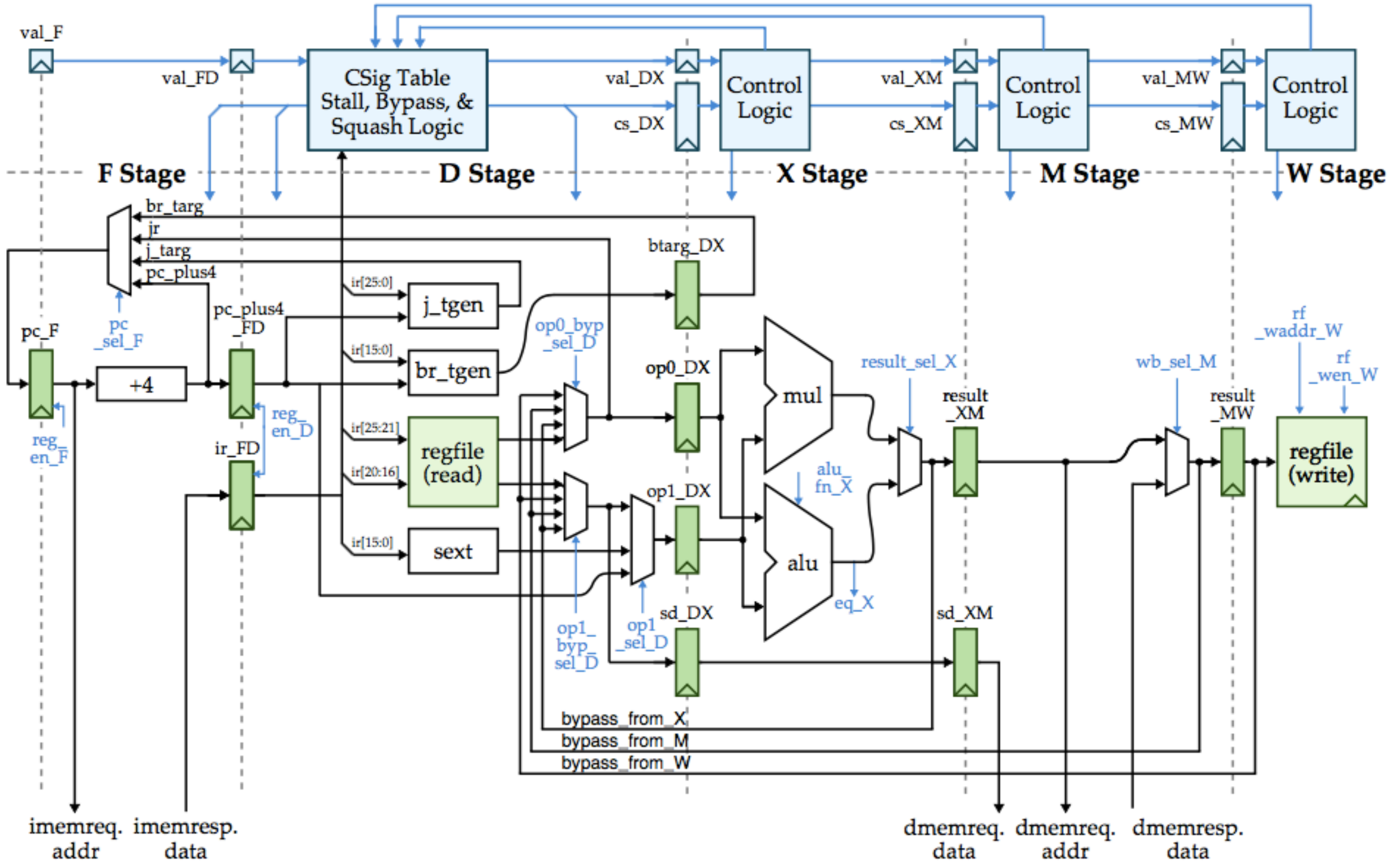
Hardware Speculation

Hardware guesses which way the control flow will go and potentially fetches incorrect instructions; detects when there is a problem and re-executes instructions the instructions that are along the correct control flow. For now, we will only consider a simple branch prediction scheme where the hardware always predicts not taken.

Pipeline diagram when branch is taken

`addiu r1, r0, 1`														
`j foo`														
`opA`														
`addiu r2, r3, 1`														
`bne r0, r1, bar`														
`opC`														
`opD`														
`addiu r4, r5, 1`														

Datapath/Control to Support Hardware Speculation



Pipeline Hazards: Structural Hazards

Structural hazards occur when an instruction in the pipeline needs a resource being used by another instruction in the pipeline. The PARCv1 processor pipeline is specifically designed to avoid any structural hazards.

Let's introduce a structural hazard by allowing ADDU, ADDIU, MUL, and JAL instructions to write to the register file in the M stage instead of waiting until the W stage. We would need to add another writeback mux in the W stage and carefully handle bypassing.

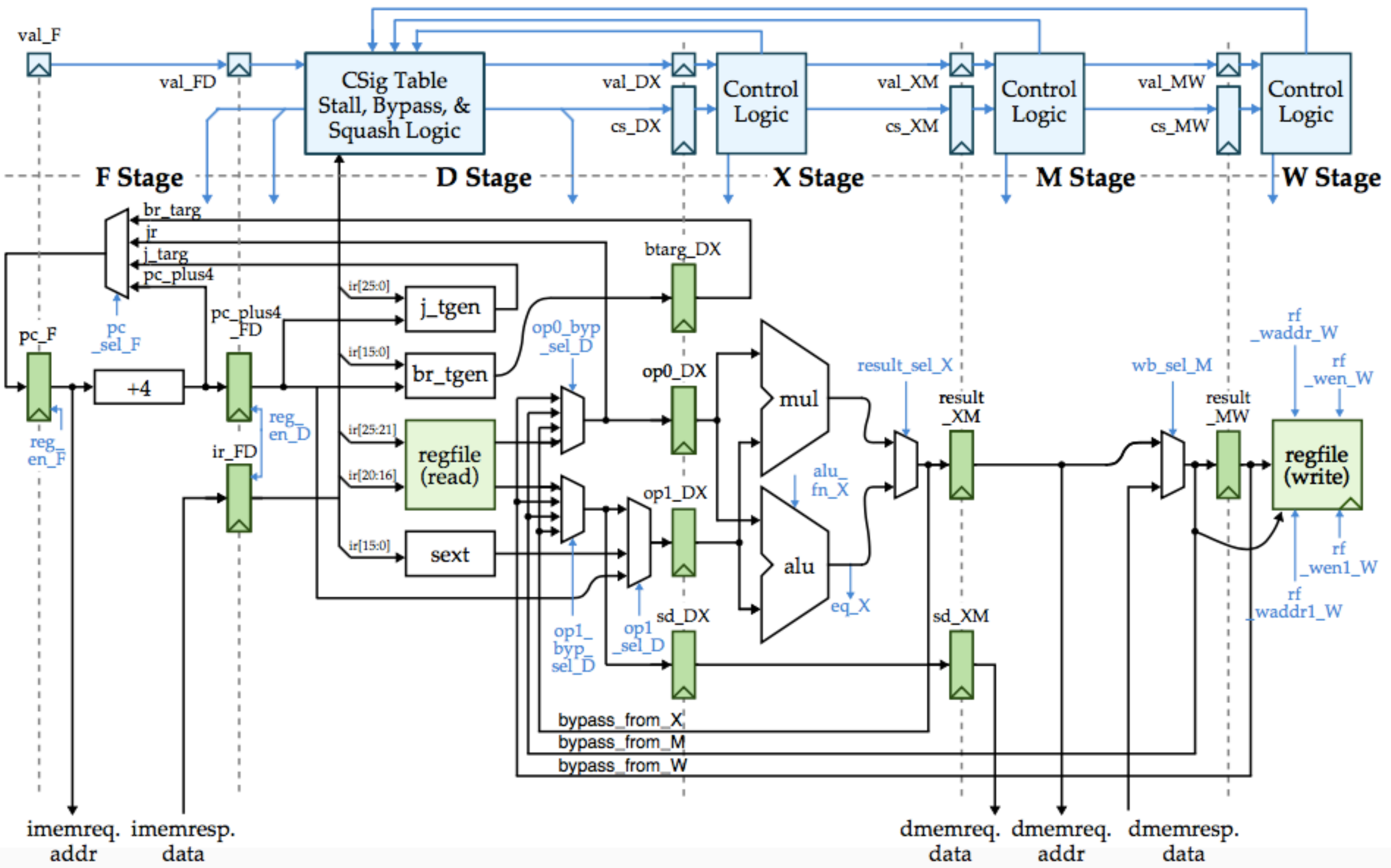
`addiu r1, r2, 1`														
`addiu r3, r4, 1`														
`lw    r5, 0(r6)`														
`addiu r7, r8, 1`														

Approaches to Resolving Structural Hazards

- Expose in ISA
  - expose control hazards in ISA forcing compiler to explicitly avoid scheduling instructions that would create hazards (i.e., software scheduling for correctness)
- Hardware stalling
  - hardware includes control logic that freezes later instructions until earlier instruction has finished producing data value; software scheduling can still be used to avoid stalling (i.e. software scheduling for performance)
- Hardware duplication
  - add more hardware so that each instruction can access separate resources at the same time

Hardware Duplication

Add a second write port so that an ADDU, ADDIU, MUL, or JAL instruction can writeback to the register file at the same time as a LW.



Pipeline Hazards: WAW and WAR Name Hazards

WAW dependencies occur when an instruction overwrites a register than an earlier instruction has already written. WAR dependencies occur when an instruction writes a register than an earlier instruction needs to read. We use architectural dependency arrows to illustrate WAW and WAR dependencies in assembly code sequences.

```
mul   r1, r2, r3

addiu r4, r6, 1

addiu r1, r5, 1
```

Introduce WAW Hazards with Iterative Multiplier

The PARCv1 processor pipeline is specifically designed to avoid any WAW or WAR name hazards. Instructions always write the registerfile in-order in the same stage, and instructions always read registers in the front of the pipeline and write registers in the back of the pipeline.

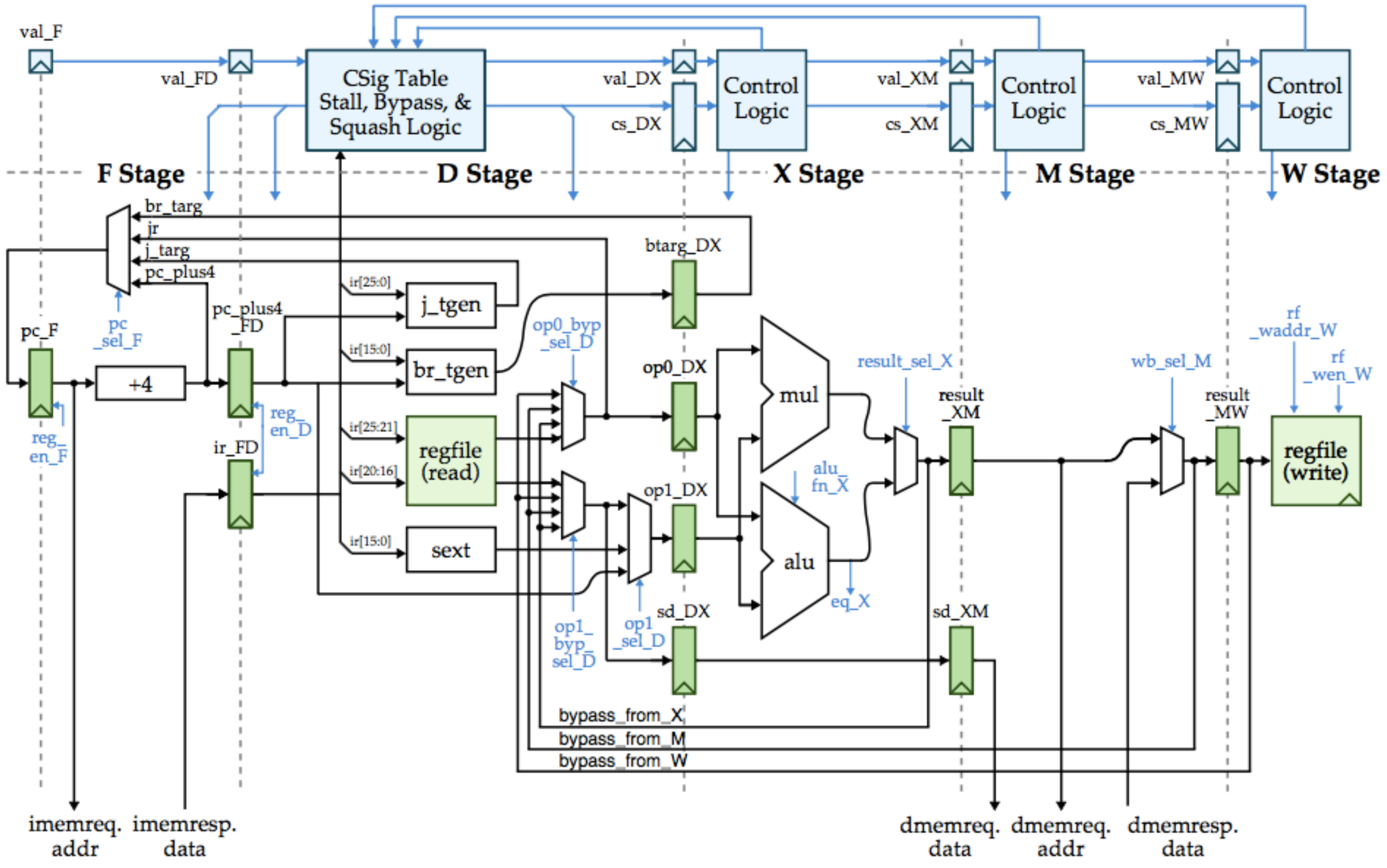
Let's introduce a WAW name hazard by using an iterative variable latency multiplier, and allowing other instructions to continue executing while the multiplier is working.

`mul   r1, r2, r3`														
`addiu r4, r6, 1`														
`addiu r1, r5, 1`														

Approaches to Resolving WAW and WAR Hazards

- Software renaming
  - programmer or compiler changes the register names to avoid creating name hazards
- Hardware renaming
  - hardware dynamically changes the register names to avoid creating name hazards
- Hardware stalling
  - hardware includes control logic that freezes later instructions until earlier instruction has finished either writing or reading the problematic register name

Estimating Cycle Time for Pipelined Processor



Questions?

Comments?

Discussion?

Acknowledgement

Cornell University, ECE 4750