Instruction Level Parallelism (ILP)

Preserve the sequential semantics of the ISA but...
try to execute as many instructions at once as we can.

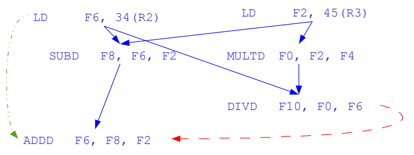
- Review of dependences
- Renaming to eliminate false dependences
- Scoreboarding: hardware out-of-order execution
- Tomasulo's Algorithm: OOO execution + renaming

Review: Dependences

- **Read-after-write (RAW)**
  - **ADD $6, $4, $5**
    ADDI $7, $6, 2
  - **Also known as a "flow dependence"**
  - **Also known as a "true dependence"**
- **Write-after-read (WAR)**
  - **ADD $6, $4 , $5**
    ADDI $4, $4, 2
  - **Also known as an "anti-dependence"**
- **Write-after-write (WAW)**
  - **ADD $6, $4, $5**
    ...
    ADD $6, $4, $5
- **WAR and WAW are "false dependences"**
  - **The dependences have to do with names, not values**
  - **They can be eliminated by "re-writing the code"**

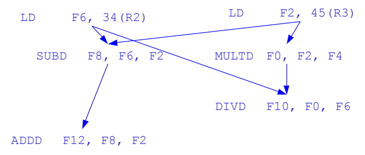
Example Code

```
LD     F6, 34(R2)
LD     F2, 45(R3)
MULTD  F0, F2, F4
SUBD   F8, F6, F2
DIVD   F10, F0, F6
ADDD   F6, F8, F2
```



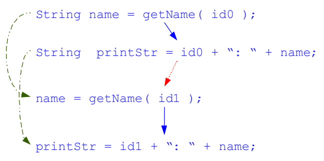
Example Code with Renaming

```
LD     F6, 34(R2)
LD     F2, 45(R3)
MULTD  F0, F2, F4
SUBD   F8, F6, F2
DIVD   F10, F0, F6
ADDD   ~~F6~~ F12, F8, F2
```



These Are Generally Applicable Ideas

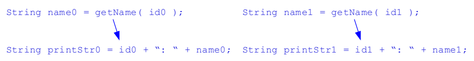
```
String name = getName( id0 );
String printStr = id0 + ": " + name;
name = getName( id1 );
printStr = id1 + ": " + name;
```



```
String name = getName( id0 );

String  printStr = id0 + ": " + name;

name = getName( id1 );

printStr = id1 + ": " + name;
```

Rewritten Code

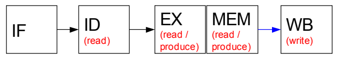
```
String name0 = getName( id0 );
String printStr0 = id0 + ": " + name0;
String name1 = getName( id1 );
String printStr1 = id1 + ": " + name1;
```



```
String name0 = getName( id0 );           String name1 = getName( id1 );

String printStr0 = id0 + ": " + name0;   String printStr1 = id1 + ": " + name1;
```

- Can't get rid of flow dependences by renaming
- Renaming costs memory
- Loops tend to produce false dependences
  - "Loop unrolling" does what that sounds like
  - Unrolled loops can benefit from renaming

Pipelining and Dependences



- Structural hazards are structurally impossible
- WAW violation is structurally impossible
- WAR violation is structurally impossible
- RAW happens

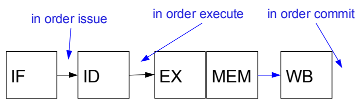
- When "things go wrong" we stall
- Stalling throws away potential performance

- *Why isn't there a hazard between MEM and IF?*

```
        jal     skip
skip:   sw      $0, 4($ra)
        lw      $0, 5($ra)      # alignment exception?
```

Instruction Level Parallelism (ILP): Pipelining

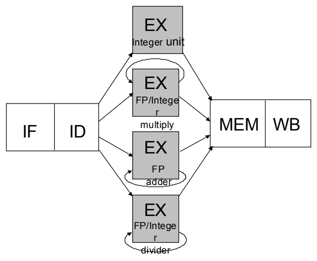
- Pipelining is a form of ILP
  - More than one instruction in flight at a time
- Respecting sequential semantics
  - Have to worry about dependences between instructions
- The *structure* of pipelines makes WAR and WAW easy
- Pipelines are:



Pipelines: Going Faster

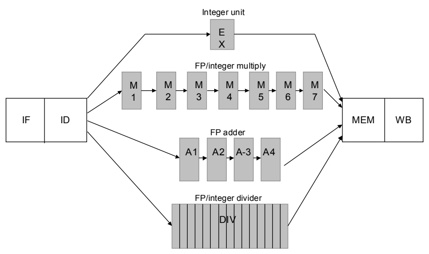
- How can we improve the performance of the 5-stage pipeline?
- We could try making the pipeline *deeper* – i.e., breaking individual stages up into multiple stages
  - Ideally, a 10-stage pipeline should support a cycle time about double that of a 5-stage, but
    - » Hazards become more costly
    - » Flushes (e.g., mispredicted branches) become more expensive
  - Diminishing returns...
- Additionally, some operations take a lot longer than others
  - For example:
    - » Cache misses...
    - » floating point is slower than integer arithmetic
  - How should we deal with that?

Floating Point and the Pipeline



The MIPS pipeline with floating-point functional units.

Pipelining the FP Units



Pipelines: Going Wide

- We have two basic choices:
  - Only one instruction may be in EX stage, no matter how long it takes it to get through there, or...
  - Let's cram instructions into EX as fast as we can
- Which should we do?
  - Reminder: We're trying to go fast...
- Putting multiple functional units in parallel is both a problem and an opportunity
- The Opportunity:
  - *Hey, this is great! Why don't I just stuff a bunch of ALUs, some memory interfaces, some float units, etc. in there?*
    - » More hardware → higher performance?
  - *In fact, why don't I issue more than one instruction per cycle?!!!*
    - » "multi-issue" → NOT part of today's material, but not far from it

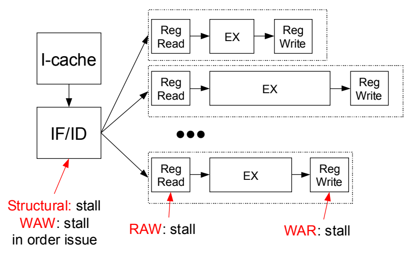
Going Wide: The Problems

- In order execution leads to under-utilization of hardware
- Parallel execution → out of order execution / completion
  - Time per stage is not a constant
    - » Structural hazards are possible
      - FP divide takes many cycles, and is not pipelined
      - May need to write more than one register in a cycle
  - Out of order execution
    - » RAW dependences may be longer
    - » "Precise exceptions" are more difficult to implement
  - Out of order completion
    - » WAR / WAW hazards are possible

Two Approaches Today

- Scoreboarding
  - **In order issue**
  - **Out of order execution**
  - **Out of order completion**
- Tomasulo's Algorithm
  - **In order issue**
  - **Out of order execution**
  - **Out of order completion**
  - **Register renaming to eliminate WAW and WAR dependences**
- **Modern processors**
  - **Descendants of Tomasulo**
  - **Add a "re-order buffer" to achieve in-order completion**
    - » **Enables precise interrupts**

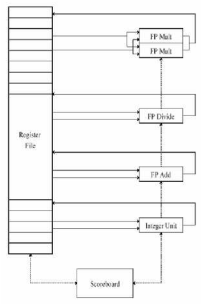
Scoreboarding



Scoreboarding

- Data path now has two largely decoupled pieces:
  - IF fetches an instruction each cycle
    - » There is a small window (buffer) of already fetched instructions
    - » If issue stalls, the buffer fills
    - » When instructions complete, they leave the buffer
  - Scoreboarding: 4-stage execution
    - » Issue – check structural /WAW hazards (stall until clear)
    - » Read ops – check RAW (wait till operands ready, read regs)
    - » Execute – execute operation. Notify scoreboard when done
    - » Write – check for WAR (stall write until clear)

Dynamic MIPS Datapath



Scoreboard Example Cycle 62

Instruction status					Read	Executi	Write
Instruction		j	k	Issue	operand	comple	Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULT	F0	F2	F4	6	9	19	20
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8	21	61	62
ADDD	F6	F8	F2	13	14	16	22

Functional unit status				dest	S1	S2	FU for j	FU for k	Fj?	Fk?
Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
0	Divide	No								

Register result status									
Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
62	FU									

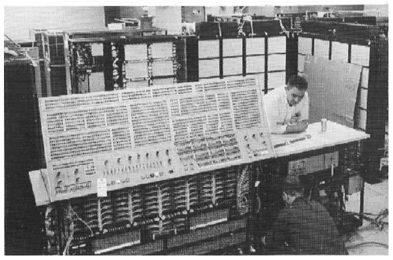
Scoreboard Summary

- Speedup 1.7 from compiler; 2.5 by hand
  - BUT slow memory (no cache)
- Limitations of 6600 scoreboard
  - No forwarding (First write register then read it)
  - Limited to instructions in basic block (small *window*)
  - Number of functional units(structural hazards)
  - Wait for WAR hazards
  - Prevent WAW hazards

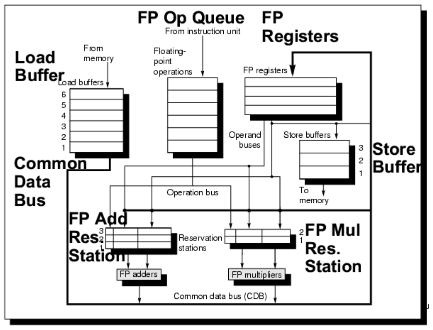
Another Dynamic Algorithm: Tomasulo Algorithm

- For IBM 360/91 about 3 years after CDC 6600 (1966)
- Goal: High Performance without special compilers
- Differences between IBM 360 & CDC 6600 ISA
  - IBM has only 2 register specifiers/instr vs. 3 in CDC 6600
  - IBM has 4 FP registers vs. 8 in CDC 6600
  - (x86 has 4 general purpose integer registers...)
- Led to Alpha 21264, HP 8000, MIPS 10000, Pentium II, PowerPC 604, …

Installation of the IBM 360/91 in the Columbia Computer Center machine room in February or March 1969



Tomasulo Organization



Tomasulo Algorithm vs. Scoreboard

- Control & buffers distributed with Function Units (FU) vs. centralized in scoreboard;
  - FU buffers called "reservation stations"; have pending operands
- Registers in instructions replaced by values or pointers to reservation stations(RS); called register renaming ;
  - avoids WAR, WAW hazards
  - More reservation stations than registers, so can do optimizations compilers can't
- Results to FU from RS, not through registers, over Common Data Bus that broadcasts results to all FUs
- Load and Stores treated as FUs with RSs as well
- Integer instructions can go past branches, allowing FP ops beyond basic block in FP queue

Three Stages of Tomasulo Algorithm

1. Issue—get instruction from FP Op Queue
   If reservation station free (no structural hazard), control issues instr & sends operands (renames registers).
2. Execution—operate on operands (EX)
   When both operands ready then execute; if not ready, watch Common Data Bus for result
3. Write result—finish execution (WB)
   Write on Common Data Bus to all awaiting units; mark reservation station available

- Where's the register renaming?
  - FU's may wait to hear a result produced by a particular other FU – that's a new name
  - Reservation stations copy the operand values – that's new names

Tomasulo Example Cycle 57

Instruction status					Execution	Write		
Instruction		*j*	*k*	*Issue*	*complete*	*Result*		Busy	Address
LD	F6	34+	R2	1	3	4	Load1	No	
LD	F2	45+	R3	2	4	5	Load2	No	
MULT	F0	F2	F4	3	15	16	Load3	No	
SUBD	F8	F6	F2	4	7	8			
DIVD	F10	F0	F6	5	56	57			
ADDD	F6	F8	F2	6	10	11			

Reservation Stations				*S1*	*S2*	*RS for j*	*RS for k*
Time	*Name*	*Busy*	*Op*	*Vj*	*Vk*	*Qj*	*Qk*
0	Add1	No					
0	Add2	No					
	Add3	No					
0	Mult1	No					
0	Mult2	No					

Register result status									
Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	…	*F30*
57	*FU*	M*F4	M(45+ R3)		(M–M)+ M()	M()–M()	M*F4/M			

- Again, in-order issue, out-of-order execution, completion

Tomasulo v. Scoreboard (IBM 360/91 v. CDC 6600)

Pipelined Functional Units	Multiple Functional Units
(6 load, 3 store, 3 +, 2 x/÷)	(1 load/store, 1 + , 2 x, 1 ÷)
window size: ≤ 14 instructions	≤ 5 instructions
No issue on structural hazard	same
WAR: renaming avoids	stall completion
WAW: renaming avoids	stall completion
Broadcast results from FU	Write/read registers
Control: reservation stations	central scoreboard