Lecture 4 & 5
Different Kinds of Parallelism

- Dependencies
- Instruction Level Parallelism
- Task Parallelism (L5)
- Data Parallelism (L5)

1

Parallelism at Different Levels

- CPU
  - Instruction level parallelism (pipelining)
  - Vector unit
  - Multiple cores
    - Multiple threads or processes
- Computer
  - Multiple CPUs
  - Co-processors (GPUs, FPGAs, …)
- Network
  - Tightly integrated network of computers (supercomputer)
  - Loosely integrated network of computers (distributed computing)

2

Dependencies

3

Correctness

- Parallel program needs to produce the same result as if it was executed on a sequential machine
- Parallel Program need to preserve program correctness
- Main hazards to correctness: Dependencies
  - Action B depends on action A; if executed out of order the program wouldn't be correct anymore.

4

Dependencies

- **Data (true) dependence**:
  An instruction j is data dependent on instruction i if either of the following holds:
  - Instruction i produces a result that may be used by instruction j, or
  - Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i (dependency is transitive)

```
i: A[1]=5;
j: B=A[1];
```

5

Dependencies Cont'd

- **Anti dependence**
  An instruction j is anti dependent on instruction i when j produces a memory location that i consumes

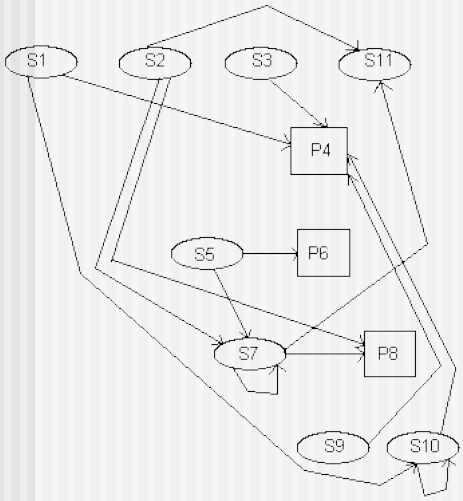
```
i: B=A[1];
j: A[1]=5;
```

- **Output dependence**
  Both i and j write to the same location

```
i: A[1]=x;
j: A[1]=y;
```

6

Dependence Graph




```
S1.   read i
S2.   sum = 0
S3.   done = 0
P4.   while i <= 5 and !done do
S5.      read j
P6.      if j >= 0 then
S7.         sum = sum + j
P8.         if sum > 100 then
S9.            done = 1
            else
S10.           i = i + 1
         endif
       endwhile
S11.  print sum
```

Loop carried dependencies are dependencies between different loop iterations, e.g. S10->P4

7

Dependencies Cont'd

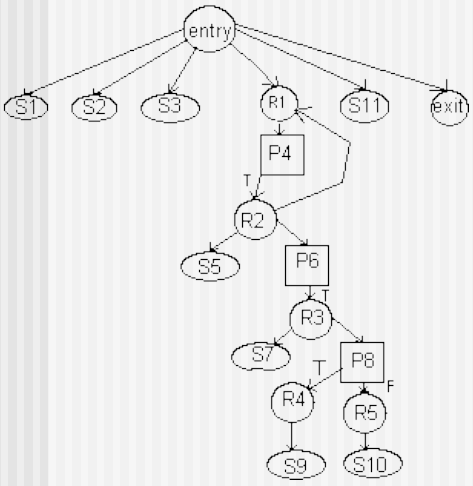
- Control dependence
  - Determines ordering of instructions with respect to a branch instruction

```
if p1 {
     S1;
};
if p2 {
     S2;
};
```

  - S1 is control dependent on p1, and S2 is control dependent on p2 but not on p1

8

Control Dependence Graph




```
S1.   read i
S2.   sum = 0
S3.   done = 0
P4.   while i <= 5 and !done do
S5.      read j
P6.      if j >= 0 then
S7.         sum = sum + j
P8.         if sum > 100 then
S9.            done = 1
            else
S10.           i = i + 1
         endif
      endwhile
S11.  print sum
```

9

Impact of Dependencies

- Braking a dependence might result in incorrect program behavior
  - Race conditions
- Hazards:
  - RAW (read after write): j tries to read a source before i writes it
    - True dependence
  - WAW (write after write): j tries to wrote an operand before it is written by i
    - Output dependence
  - WAR (write after read): j tries to write a destination before it is read by i
    - Anti dependence
- Naive approach: Don't brake dependencies
  - But that doesn't leave us with a lot of parallelism in many cases

10

How to deal with Dependencies

- True dependence:
  - Synchronize
  - Transform the code
- Anti and output dependence:
  - Renaming (introduce new variable)
- Control dependence:
  - Speculative execution

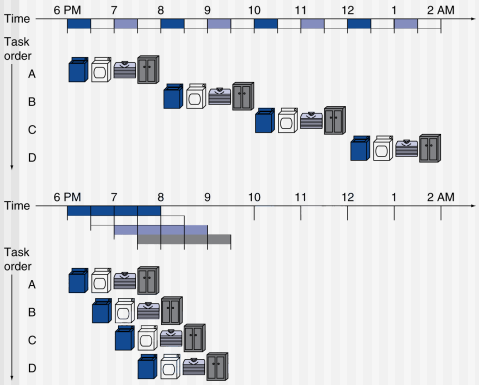
11

Instruction Level Parallelism

12

Pipelining

- Pipelining is an implementation technique in which multiple instructions are overlapped in execution
  - Almost universally used in today's processors



- Laundry analogy
  - 4 tasks: wash, dry, fold, put away
- In constant use this leads to speedup of ~4, i.e. number of stages

13

MIPS Pipeline

- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register

14

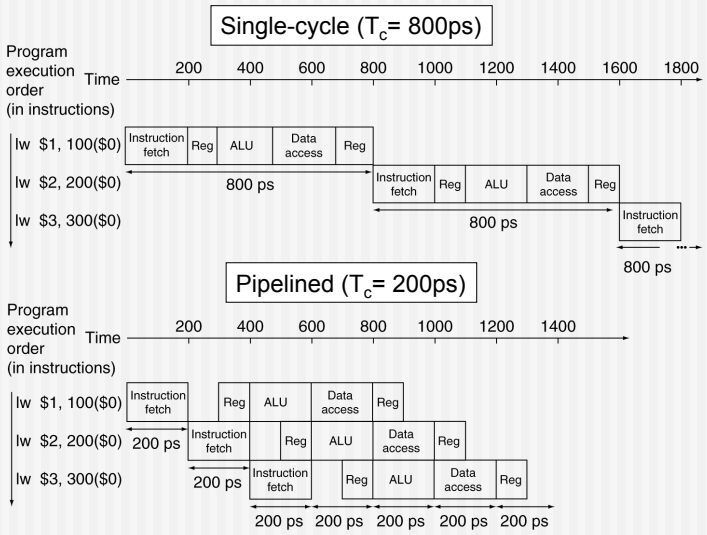
Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time (instruction latency)
Load word lw	200ps	100 ps	200ps	200ps	100 ps	800ps
Store word sw	200ps	100 ps	200ps	200ps		700ps
R-format (add, sub, AND, OR, slt)	200ps	100 ps	200ps		100 ps	600ps
Branch beq	200ps	100 ps	200ps			500ps

15

Pipeline Performance



Note that all pipeline stages take the same time

16

Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions$_{pipelined}$ = $\frac{\text{Time between instructions}_{nonpipelined}}{\text{Number of stages}}$
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

17

Good News

- In most cases the CPU hardware and compiler will be able to use pipelining efficiently
- Parallelism "for free"

18

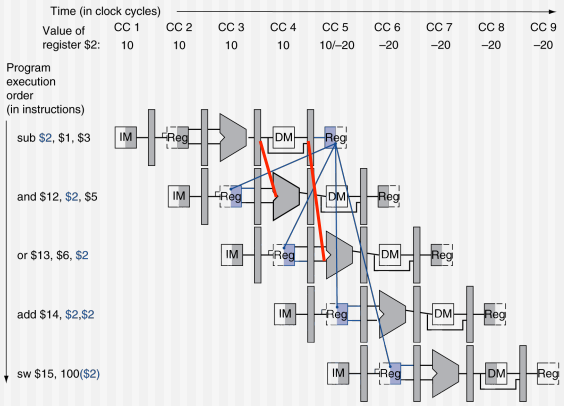
Bad News

- In some cases hardware/compiler can't do a good job
- It's important to have a basic understanding what is happening under the cover
- Issues in IPL and techniques to overcome them are in many cases also applicable to more coarse grained parallelism

19

How to deal with Hazards?

- True dependence:
  - AND and OR would get wrong data in $2
  - But data is already available before written back

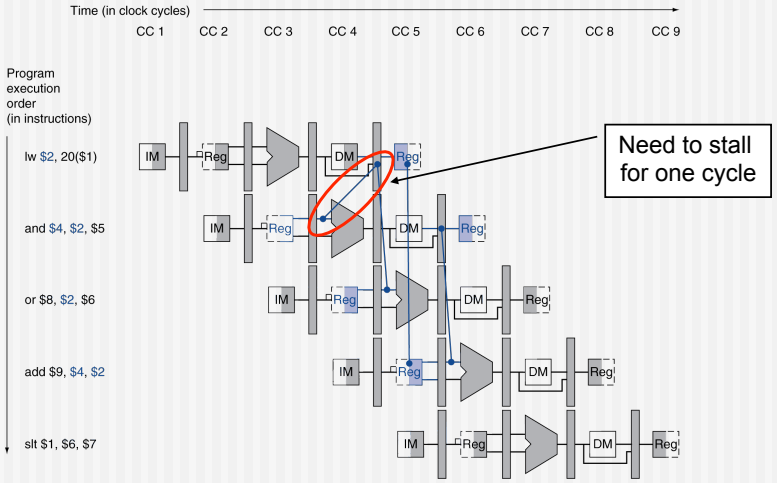


- Forwarding: hold data in pipelined registers and forward between them

20

True Dependence Cont' d

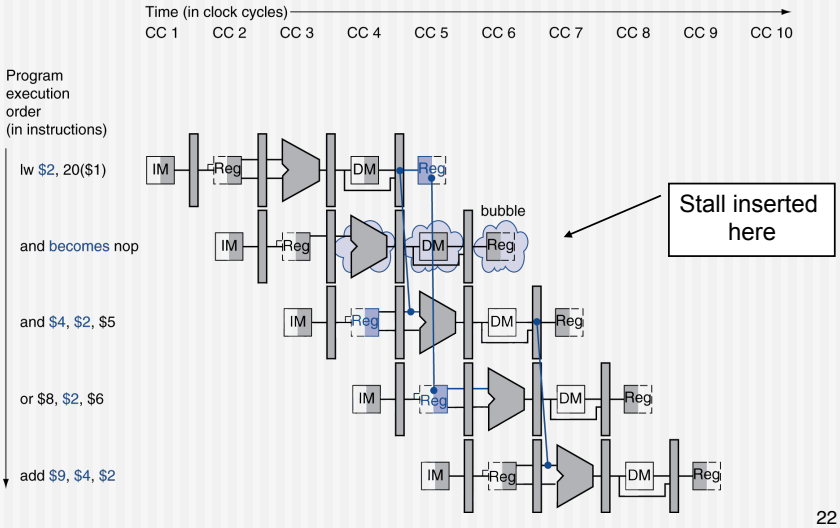
- Sometimes forwarding is not possible
  - Data doesn' t exist yet, when it would be needed



21

True Dependence Cont' d

- Add stalls ("bubbles")



22

Anti/Output Dependence

- Write after Read or Write after Write
- Can be relatively easily overcome by introducing extra registers
- Particularly useful for loop-carried anti-dependencies

```
do i=1,n
   V[i]=V[i]-2*V[i+1]
enddo
```

```
W=V
do i=1,n
   V[i]=W[i]-2*W[i+1]
enddo
```

23

Data Hazards Summary

- Data Hazards may impact the performance of pipelines
- Hardware checks typically take care of data hazards and guarantee correct program behavior
- Compiler needs to know the pipeline architecture to produce efficient code avoiding stalls

24

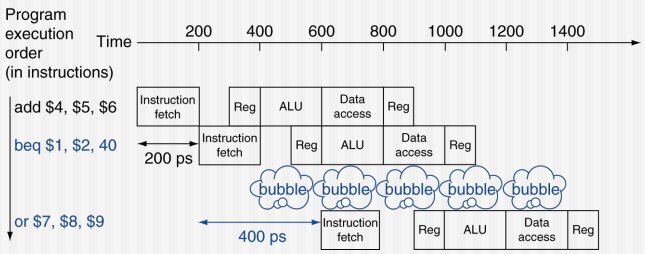
Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction
    - Still working on ID stage of branch

25

Stall on Branch

- Wait until branch outcome determined before fetching next instruction



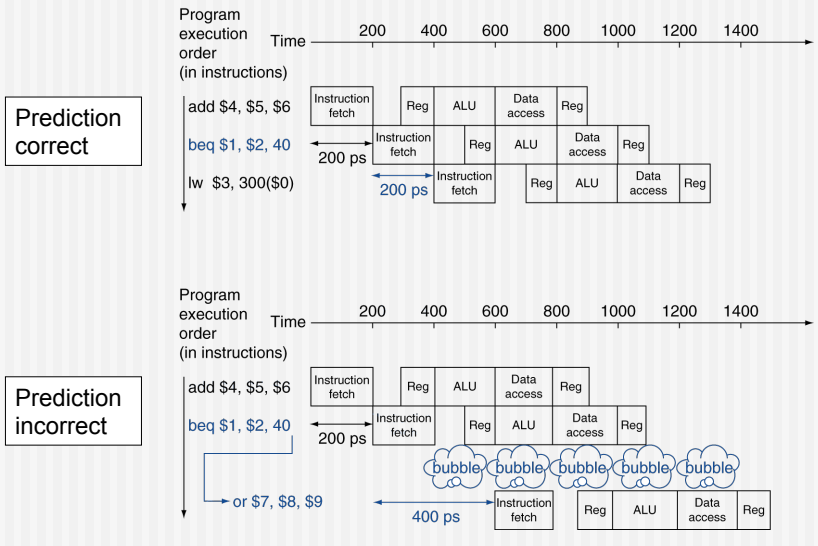
26

Branch Prediction

- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong

27

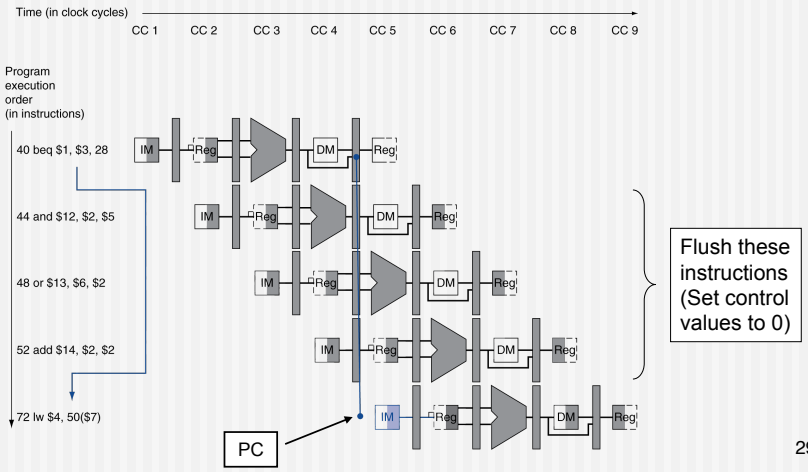
MIPS with Predict Not Taken



28

Branch Prediction Cont'd

- If branches are taken, pipeline is filled with wrong (intermediate) results that need to be flushed



29

More-Realistic Branch Prediction

- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history

30

How to increase ILP

- Increase depth of pipeline
  - More instructions can work in parallel
  - Penalty on stalls and flushes are larger
- Multiple issue
  - Replicate components so multiple instructions can be issued every pipeline stage
  - Dependencies reduce this typically
- Speculation
  - "Guess" what to do with an instruction
  - Requiress roll-back
- Loop unrolling
  - Increase the size of loop body to exploit more parallelism

31

Multiple Issue

- Static multiple issue
  - Compiler groups instructions to be issued together
  - Packages them into "issue packets"
  - Compiler detects and avoids hazards
  - Think of an issue packet as a very long instruction
    - Specifies multiple concurrent operations
    - ⇒ Very Long Instruction Word (VLIW)
- Dynamic multiple issue
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime
  - "Superscalar" processors

32

Does Multiple Issue Work?

- Yes, but not as much as we'd like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
  - e.g., pointer aliasing
- Some parallelism is hard to expose
  - Limited window size during instruction issue
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well

33

Speculation

- "Guess" what to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right
    - If so, complete the operation
    - If not, roll-back and do the right thing
- Common to static and dynamic multiple issue
- Examples
  - Speculate on branch outcome
    - Roll back if path taken is different
  - Speculate on load
    - Roll back if location is updated

34

Compiler/Hardware Speculation

- Compiler can reorder instructions
  - e.g., move load before branch
  - Can include "fix-up" instructions to recover from incorrect guess
- Hardware can look ahead for instructions to execute
  - Buffer results until it determines they are actually needed
  - Flush buffers on incorrect speculation

35

Loop Unrolling

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead
- Use different registers per replication
  - Called "register renaming"
  - Avoid loop-carried "anti-dependencies"
- Gives more freedom in scheduling instructions and typically exploits the pipeline better
  - Larger basic blocks gives more freedom to dynamic scheduling

36

Dynamic Pipeline Scheduling

- Allow the CPU to execute instructions out of order to avoid stalls
  - But commit result to registers in order
- Example

```
lw    $t0, 20($s2)
addu  $t1, $t0, $t2
sub   $s4, $s4, $t3
slti  $t5, $s4, 20
```

  - Can start sub while addu is waiting for lw

37

Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
- Not all stalls are predicable
  - e.g., cache misses
- Can't always schedule around branches
  - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards

38

Power Efficiency

- Complexity of dynamic scheduling and speculations requires power
- Multiple simpler cores may be better

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/ Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000MHz	22	3	Yes	1	75W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W
Core	2006	2930MHz	14	4	Yes	2	75W
UltraSparc III	2003	1950MHz	14	4	No	1	90W
UltraSparc T1	2005	1200MHz	6	1	No	8	70W

39

Pipelines Summary

- Pipelining improves performance by increasing instruction throughput
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
- Instruction set design affects complexity of pipeline implementation
- Multiple issue and dynamic scheduling
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall

40

Take Home Message

- Avoid dependencies
  - Or at least try to structure them
- Avoid branches
  - Increase basic block size
    - Basic block is a sequence of instructions w/o branches
- This also holds for exploiting parallelism at a higher level
  - Data and Task parallelism

41

Summary and Outlook

- Dependencies
- Pipelines
- How to increase parallelism
- Parallelism at a higher level
  - Task and Data parallelism

42

Further Readings

- **Computer Organization and Design - The Hardware/Software Interface,** 4th Edition, David A. Patterson and John L. Hennessy
  - Chapter 4 (particularly 4.5, 4.7, 4.8, 4.10)
- Expert reading:
  - **Computer Architecture - A Quantitative Approach**, 4th Edition, John L. Hennessy and David A. Patterson
    - Appendix A and Chapter 2-3

43

Exercises

13. Analyze the dependence graph given on slide 7 and add the nature of the dependence (true, anti, output) to each of the vertices. Identify loop carried dependencies and add missing vertices (are all anti-dependencies there?)

44

Exercises Cont'd

14. Consider a nonpipelined machine with 6 execution stages of lengths 50 ns, 50 ns, 60 ns, 60 ns, 50 ns, and 50 ns.
    1. Find the instruction latency on this machine.
    2. How much time does it take to execute 100 instructions?
    3. Suppose we introduce pipelining on this machine. Assume that when introducing pipelining, the clock skew adds 5ns of overhead to each execution stage.
       - What is the instruction latency on the pipelined machine?
       - How much time does it take to execute 100 instructions?
       - What is the speedup obtained from pipelining?

45

Exercises Cont'd

15. Consider the following loop:

```
do i=1,100
  if (i mod 2 == 0) then
     a(i) = x
   else
     a(i) = y

enddo
```

    1. What effect could loop unrolling have to this loop? Describe both loop overhead and loop body
    2. What would be a good first unrolling depth?
    3. Provide unrolled code

46