Topic 2: Advanced Pipelining

- **Instruction-Level Parallelism:**
- The potential overlap among instructions
- Want to increase the amount of parallelism exploited among instructions

Dynamic Hardware Scheduling.

- **Hardware schemes for getting parallelism from instructions**
- **Idea:**
    - **Decode instructions, determine dependencies**
    - **Keep track of what operands are needed**
    - **As soon as operands become available, let instructions start execution**
    - **Fetch instructions in order**
    - **But they may execute out-of-order**

- **We will look at two mechanisms: Scoreboarding and Tomasulo's algorithm**
  Both ideas date from the 1960s
  Were originally used in supercomputers
  Now available in desktop microprocessors

- **Dynamic scheduling of instructions with hardware**
    - **Goal: reduce the number of stalls**

  Can handle some cases where dependencies are unknown at compile time
  (e.g., involve a memory access)
  Simplifies compiler
  Allows efficient execution on different pipelines

- **Disadvantage of pipeline schemes (up to now)**
- In-order instruction issue
- If dependencies, then stall
- Example: DIVD F0, F2, F4
  ADDD F10, F0, F8
  SUBD F8, F8, F14
- The SUBD instr gets stalled in the pipeline because ADDD is stalled

- But, not data dependent
- Could execute if out-of-order execution were allowed

- **Idea: Separate old instruction decode into two logical components**

1. Checking for structural hazards
2. Waiting for resolution of data hazards

- Want to let instruction execute as soon as operands are ready
- Still check for structural hazards before sending to an execution unit, so
  - **In-Order Instruction Issue**
- But,execute as soon as operands are available, so may not execute in order:
  - **Out-of-Order Completion**

- As in section on FP and interrupts,
  - makes handling exceptions difficult
- These early machines (CDC6600) had imprecise exceptions
- Will discuss precise exceptions later

- **Effectively, split ID phase into two phases**

1. ISSUE: decode instr, check for structural hazards

2. READ OPERANDS: wait for data hazards to be resolved, then read operands

- These two stages followed by EX
- Will possibly take multiple cycles

SCOREBOARDING.

- **Technique for allowing instructions to execute out-of-order when there are sufficient hardware resources and no data dependencies**
- Scheme shown in book is simplified version of CDC6600

- **First, show transparency with data structures on it!!**

- **Scoreboard needs to track:**

1. **Instruction status (which of 4 steps)**

   Issue

   Read Operands

   Execution complete

   Write results

2. **Functional unit status**

   Busy

   Operation

   Fi -- destination register

   Fj, Fk -- source registers

   Qj, Qk -- if operands j, k are not ready, indicates which func. unit will generate result

   Rj, Rk -- ready flags; indicate that operands are ready;
   when all operands are ready, execution may proceed

3. **Register result status**

   Indicates which functional unit will write a register

- **Four-step execution (replaces ID, EX, WB in DLX pipeline)**

1. **ISSUE**

   Checks for structural hazard: Is functional unit free?

   Checks for WAW hazards:

   No other active instruction has same destination register?

   If a hazard, stall -- no further instruction issue

   (Remember, issue in order!)

   If no hazard, issue and update data structures

2. **READ OPERANDS**

   Checks for RAW hazards

   Scoreboard monitors availability of operands

Hardware does no “forwarding”

When operands are ready, tells functional units to

read operands and

begin execution

3. EXECUTION

Operate on operands

May take multiple cycles

When execution complete, notifies scoreboard

4. WRITE RESULT

After execution is complete

Checks for WAR hazards

If none, writes result

If WAR, stall instruction until resolved

- **Execution units assumed:**

  2 FP MULT

  1 FP ADD

  1 FP DIV

  1 integer unit

- **Integer unit used for ALL:**
  - **Memory references**
  - **Branches**
  - **Integer ALU ops**

- **Assume for functional units:**

  FP ADD takes 2 cycles

  FP MULT takes 10 cycles

  FP DIV takes 20 cycles

- **Scoreboard Example**

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

Tomasulo's Algorithm

- **Another Hardware scheme for getting parallelism from instructions**
- **Tomasulo's Algorithm**
- Based on the IBM 360/91
- Originally designed to overcome
  - Fixed architecture, with only small number of floating point registers and
  - Long memory access and floating point delays

- **Combines key elements of scoreboarding with REGISTER RENAMING**
- To avoid WAR and WAW hazards
- **Main differences from scoreboarding:**
  - **Decentralized control (hazard detection and execution control)**
  - **Reservation stations hold operands at functional units,**
  - **So don't need to read from register file**
  - **Common data bus allows functional units to read operands as they pass by**
  - **Don't contend for register file**
- Use RESERVATION STATIONS to rename registers
- Buffer operands of instructions waiting to issue -- as soon as operands are ready
- Don't issue instructions until all operands are ready
- May be more reservation stations than real registers
- So, can elminate hazards that could not be eliminated by a compiler

- Draw Figure 4.8: Load buffers, floating point operation queue, floating point registers, store buffers, reservation stations, FP adders and multipliers
- Steps of instruction execution

1. **Issue: Get an instruction from floating point operation queue**

   If a FP operation, issue if empty reservation station

   AND send operands to reservation station if they are in registers

   If Load/Store, can issue if an available buffer

   Otherwise, have a structural hazard: must stall

   Performed renaming of registers by sending operands to reservation stations

2. **Execute: If one or more of operands not yet available, monitor the CDB**

   When operand becomes available, it is stored in appropriate reservation station

When both operands available, execute

Checks for RAW hazards

3. **Write result: when result available, write on CDB and from there into any registers or functional units waiting for the result**

- **Differences from Scoreboarding:**

1. No checking for WAR or WAW: eliminated when register operands are renamed
2. CDB broadcasts results; don't wait for registers to be written
3. Loads and stores are treated as basic functional units

- **Control data structures: reservation stations, register file and load/store buffers**
- Tag fields on everything (except load buffers): names for extended set of virtual registers used in renaming
- **As discuss, write Status registers on Board**
- **Instruction Status:**
- Remember, not a centralized structure as in Scoreboarding; we just use it to keep track!
    - Issue
    - Execution Complete
    - Write Result
- **Reservation Stations**
    - Assumes three adder units, two multiplier units
    - Unit Name
    - Busy Flag
    - Operation
    - Vj, Vk: Values of operands (copy operand values to reservation station!)
    - Qj, Qk: Units from which the results will come
- **Load/Store Unit Status**
    - Assumes 6 load units and 3 store units
    - Busy Flag
    - Address
    - For store unit: Qj flag--where will result come from
- **Register Result Status**
    - Keeps track of : either the value in register or the unit that will load the register

- Go through entire example
- **Review advantages of Tomasulo's Algorithm:**

1. Distribution of hazard detection and control logic

   (Because of distributed reservation stations and CDB:

   Multiple instructions waiting on a single operand can all start execution as soon as the operand is broadcast on the common data bus)

2. Elimination of stalls for WAW and WAR hazards

   (Because of register renaming and storing operands into reservation stations as soon as available)

- **Next Topic: Dynamic Hardware Branch Prediction**
- Scoreboarding, Tomasulo's Algorithm: deal dynamically with data hazards (not in compiler)
- Now we look at dynamic approaches to dealing with control hazards (branches)
- Previous schemes (filling branch delay slots) made static guesses about branch outcomes
- We can do better if we monitor branch outcomes during execution and guess then
- Need accurate prediction scheme with low penalty for incorrect prediction

- Goal: Allow processor to resolve outcome of a branch early
- Prevent control dependencies from causing stalls

1. **Branch Prediction Buffer**

   Simplest scheme; Also called BRANCY HISTORY TABLE

   Idea: Record result (taken or not taken) of last time each branch was executed

   On next execution of same branch, predict the same outcome as last time

   Details: (Branch addr (4 bits) ------ access long column of entries, extract prediction)

   Access table using bits of branch instruction

   To limit size of prediction buffer, use the lower bits of the address

   Table is effectively a direct-mapped cache

   If two branch instructions share the same lower bits, may access history of a different branch (not necessarily the current one)

   Doesn't matter--just make that the prediction

- How effective is this scheme?

  Not very, since not saving much history

  On any loop, will have two mispredictions in a row

(on first entering the loop and finally leaving it)

Example: say execute loop nine times in a row, then don't take it

First time through loop, will predict not taken, but will actually take

Then for next eight times, will correct predictly that loop is taken

When leaving loop, will predict taken but not taken

So, 80% of the time we predict correctly, but branch is taken 90% of time

- Limitation is only keeping one bit of history
- Better: use two bits of prediction
- State diagram: 2-bit predictor, show state transitions
    - 0 = not taken, 1 = taken
    - 00 = not taken twice: predict not taken
    - 01 = taken last time, but not time before: predict not taken
    - 10 = not taken last time, but taken time before: predict taken
    - 11 = taken twice: predict taken
- **Can generalize this scheme to n bits**
- If counter > 2^n -1 (halfway), then predict taken; if correct, increment, if incorrect, decrement
- But, performance of 2 bits is as good as n bits
- **Figure 4.14 in book: prediction accuracy of 4096-entry two-bit prediction buffer on SPEC89 benchmarks**
- Average 11% mispredictions for integers, 4% floating point
- 0% on matrix 300: scientific, loop intensive
- worst is 18% on eqntott
- Figure 4.15 in book: 4096-entry buffer approximately as good as infinite size buffer

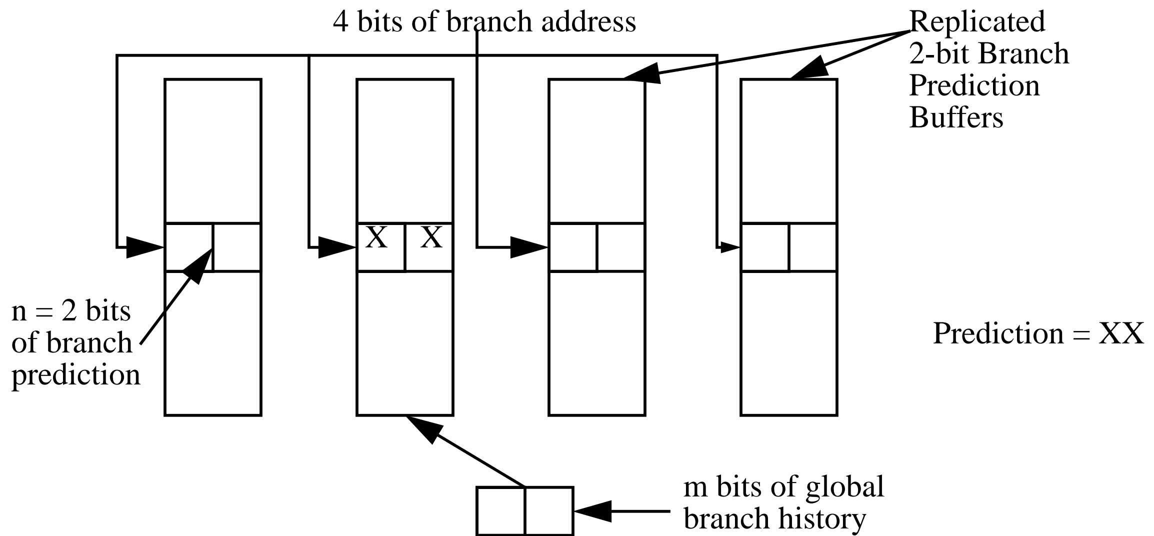
2. Correlation

- Don't just use the recent history of this branch
- Also look at behavior of other branches
- Example:

```
if (aa==2)
    aa=0;
if (bb==2)
    bb=0;
```

if (aa!=bb){ ...

- Three branches, third is correlated with first two
- If both taken, then third won't be taken
- **Correlating predictors or two-level predictors**
- (m,n) predictor: uses m bits of global branch history (last m branches taken/not taken)
- Create 2^m branch predictors of n bits each
- Replicating n-bit branch prediction buffers
- Example (2,2) predictor

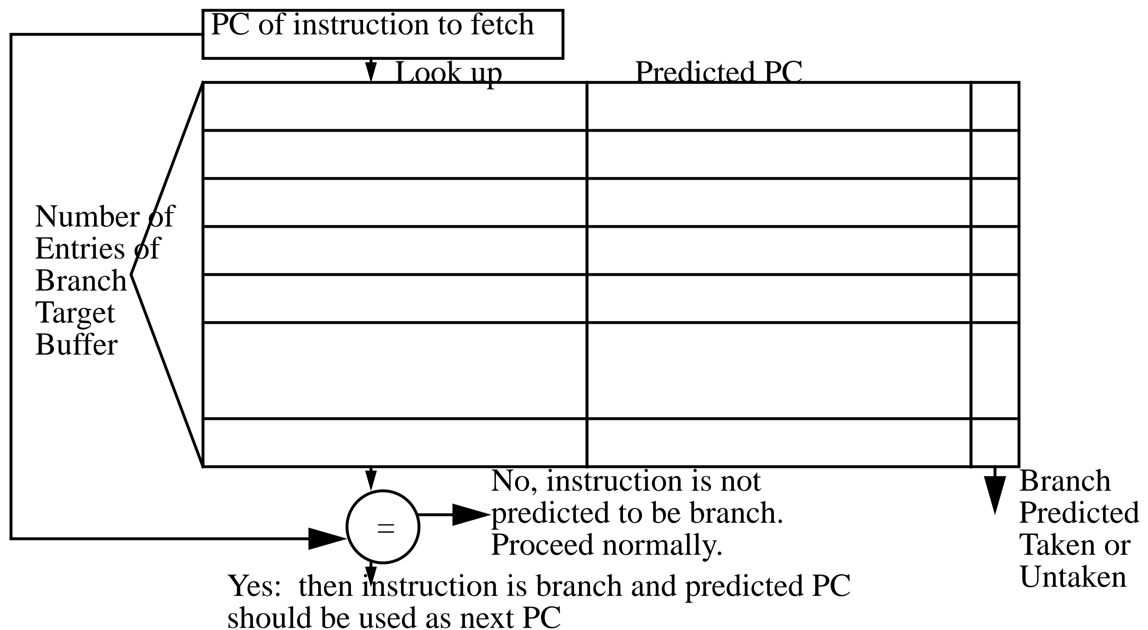


- Performance: Figure 4.21
- Compares a 4096-entry, 2-bit branch prediction buffer with a 1024-entry, correlated, (2,2) predictor
- Correlated predictor performs better in some cases (e.g., eqntott down to 6% from 18% misprediction)

3. Branch Target Buffers

- Other schemes gave us branch prediction early
- Also need branch destination early: DLX, want next instruction by end of IF
- Use address (PC) of branch to access table with predicted PC of branch and prediction (taken/not taken)
- Effectively a branch prediction cache,
    - stores predicted address for next instruction after the branch
    - stores prediction (taken or untaken)

- Sending out new address before instruction is decoded; so, must know whether predicted as a taken branch; if we get a hit, then we assume it is a branch and use the predicted address if branch predicted taken
- On DLX: access during IF cycle, then on next cycle, fetch address of possible branch
- If get a hit in BTB, know the predicted address at the end of IF cycle



- We have to check a match on the whole PC, not just lower address bits
- Because will immediately fetch if we predict it is a taken branch
- Sometimes, will mispredict: then have to abort fetched instruction, fetch new instruction and update the prediction in the BTB
- No stalls if entry in BTB and correctly predict a taken branch (or no branch!)
- May want to stall both taken and untaken PCs, but only need to store branch target
- If wrong, at least a 2-cycle penalty: halt instructions, rewrite buffer entry
- IF: Send PC to memory and branch-target buffer
- Entry found in BTB?
- If no, on ID: is instructino a taken branch?
    - If no, normal instruction execution
    - If yes, enter branch PC and next PC into branch target buffer, abort execution of old instruction
- If yes (entry in branch-target buffer), send out predicted PC on ID phase
- Also on ID: is branch taken? Yes, then branch was correctly predicted (no penalty)
- If no, then Misprediction! Kill fetched instruction, restart fetch at other target; delete entry from target buffer

Today: Loop Unrolling to increase parallelism.

- How do we use the compiler to increase ILP?

- **We have already looked at compiler scheduling: rearranging the code**
- **Hard to do it within basic blocks of code:**
- Example: GCC has 17% control transfer instructions: 5 instructions + branch instr
- Basic blocks small, instructions highly dependent

- **Want to increase ILP, must do scheduling across basic blocks**
- **Potentially a lot of parallelism between different iteractions of loops**

  Example: for (i=1; i< 1000; i=i+1)

  x[i] = x[i] + y[i];

  Completely parallel

- **Loop Unrolling**
  - Compiler schedules code for multiple iterations of loop, removing branches between iterations
- **Loop Unrolling Example (p. 225-227)**

```
for (i=1; i<=1000; i++)
    x[i] = x[i] + s;
```

- DLX code:

```
Loop: LD   F0, 0(R1)  ; get next vector element
    ADDD F4, F0, F2;  add scalar in F2
    SD  0(R1), F4;     store result
    SUBI R1, R1, 8;    decrement pointer 8 bytes (per double word)
    BNEZ  R1, Loop
```

  Assume that R1 is the address of the lst element of the vector--count down

  Simplicity: assume array starts at location 0

- Assumptions about floating point latencies (don't correspond to DLX, but similar)

TABLE 1.

Producer of Result	Consumer of result	Latency
FP ALU operation	Another FP ALU operation	3
FP ALU operation	Store Double	2
Load Double	FP ALU operation	1
Load Double	Store Double	0 (result of load can be used without stalling store)

- Assume functional units are fully pipelined and replicated
- Assume no strutural hazards
- Can always issue new instructions
- Assume standard DLX integer pipeline: so branches have a dela of one cycle
- How long does it take to execute our loop?

TABLE 2.

Cycle	Instruction
1	Loop: LD F0, 0(R1)
2	Stall
3	ADDD F4, F0, F2
4	Stall
5	Stall
6	SD 0(R1), F4
7	SUBI R1, R1, #8
8	BNEZ R1, LOOP
9	Stall (branch delay)

- Scheduling/reordering
- How might we change the order of instructions (schedule) to minimize number of stalls?

TABLE 3.

Cycle	Instruction
1	Loop: LD F0, 0(R1)
2	stall
3	ADDD F4, F0, F2
4	SUBI R1, R1, #8 <---- decrement first
5	BNEZ R1, Loop
6	SD 8(R1), F4 <--- branch delay (already decremented, so add back!)

- 6 total cycles; of these, three are branch overhead

- **Unroll the Loop**
- Increase the ratio of useful instructions to loop overhead
- Last time: loop = 6 instructions, only 3 (LD, ADDD, SD) did work of loop
- Others overhead
- Unroll our loop 4 times

- Assumes R1 is a multiple of 4

TABLE 4.

Cycle	Instruction
1	Loop: LD F0, 0(R1)
2	Stall
3	ADDD F4, F0, F2
4	Stall
5	Stall
6	SD 0(R1), F4 <--- (drop SUBI and BNEZ)
7	LD F6, -8(R1)
9	ADDD F8, F6, F2
12	SD -8(R1), F8
13	LD
15	ADDD
18	SD
19	LD
21	ADDD
24	SD
25	SUBI
26	BNEZ
27	(branch delay)

- 4 loops in 27 cycles (unscheduled): 6.8 cycles per loop vs. 9 for unscheduled before
- Removed the separate decrement instruction
- Only include branch instrution at end of unrolling

- **Now schedule this loop:**

TABLE 5.

Cycle	Instruction
1	Loop: LD F0, 0(R1) (assumes no structural hazard--issue a load every cycle)
2	LD F6, -8(R1)
3	LD F10, -16(R1)
4	LD F14, -24(R1)
5	ADDD F4, F0, F2
6	ADDD F8, F6, F2
7	ADDD F12, F10, F2
8	ADDD F16, F14, F2
9	SD 0(R1), F4
10	SD -8(R1), F8

TABLE 5.

Cycle	Instruction
11	SD -8(R1), F8
12	SUBI R1, R1, #32 (do decrement now so that it can be used in the branch check)
13	BNEZ R1, Loop
14	SD 8(R1), F16 (8 - 32 = -24) (want to schedule something in the looop delay)

- **4 loops in 14 cycles: 3.5 cycles per iteration**
- **3 instructions do useful work!!**
- **Assumptions we made:**
- OK to move last store past SUBI
- OK to move loads before stores: check that get right data
  Compilers need complex rules to determine dependencies!!

Parallel instructions: definition

- can execute simultaneously in a pipeline wihtout any stalls,
- assuming no structural hazards

- **Dependent instructions: not parallel**
- **Dependencies are properties of programs; whether they result in hazards depends on pipeline organization**

- Types of dependence:

1. **Data dependence (corresponds to RAW hazard for hardware)**
   Instr j depends on instr i if:
   1. i produces a result used by j
   2. j depends on k,k depends on i (chained dependence)
      Easy to determine for registers
      Harder for memory: does 100(R4) equal 20(R6) -- how can compiler know?
      For different loop iterations, does 20(R6) = 20(R6)?

2. **Name dependence**
   instr i and j use same name (reg or memory location)
   1. antidependence -- j writes location that i reads from (WAR equivalent)
   2. output dependence -- i and j write same location (WAW hazard)
   Register instructions can execute in parallel

May be able to re-allocate registers to avoid conflicts

Memory: again, harder: same examples as before -- how to know if addresses same?

One solvable problem: if R1 doesn't change over a code sequence, then:

0(R1), -8(R1), -16(R1), -24(R1) are all distinct

Can move them around

We will use this in loop unrolling!

3. Control dependence: branches

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

S1 control dependent on p1; S2 control dependent on p2, but not on p1

Obvious constraints:

1. if i depends on a branch, can't move it before branch so it is no longer controlled by branch
2. if i does not dependon a branch, can't move it after branch so that its execution depends on branch

Example:

```
          ADD R1, R2, R3
          BEQZ R12, skipnext
          SUB R4, R5, R6
skipnext: OR R7, R1, R9
          MULT R7, R1, R4
```

What are control (and data) dependencies?

SUB is control dependent on branch -- only done if branch is NOT taken

ADD can't be moved immediately after the branch

OR is data dependent on the ADD (R1)

Can move OR before the branch but not before the ADD

MULT and OR are name-dependent (WAW)

- **Rearranging instructions Must not affect correctness**
- **Two probabilities critical to correctness:**

1. Exception behavior

    Can't make changes in execution order that change how exceptions are raised

    At least, can't generate new exceptions

    Example:

```
      BEQZ   R2, L1
      LW    R1, 0(R2)
L1: .......
```

Accessing 0(R2) may cauase memory protection violation

Can't move before branch -- > not a data dependency

If branch taken, would never get an exception

Control dependency keeps us from moving, so it's OK

2. Data Flow

```
      ADD R1, R2, R3
      BEQZ R4, L
      SUB R1, R5, R6
L1:  OR   R7, R1, R8
```

OR statement: R1 value depends on whether branch is taken or not

Not just data dependent

Control dependencies prevent moving SUB instruction, so OK

(Later in chapter: speculative execution will let us change control dependencies but maintain data flow)

- **Loop level analysis:**
- **Determine dependencies among operatnds across iterations of the loops**
- **When is it safe to unroll a loop?**
- **Detect data dependencies**

```
for (i=1; i<100; i++){
      A[i+1] = A[i] + C[i];           /* S1 */
      B[i+1] = B[i] + A[i+1];         /* S2 */
}
```

1. S1 uses value computed by S1 on earlier iteration

   iteration i computes A[i+1]

   iteration i+1 uses A[i+1]

   Same for S2 for B[i+1]

   - **Loop-carried dependence!!**
   - **Between iterations**
   - **Successive iterations of S1 must execute in order, not in paralle!**

2. S2 uses value A[i+1] calculated in S1 ---->> RAW hazard

- **Multiple Issue Machines: Superscalar and VLIW**
- Last techniques: Eliminate data and control stalls, try to achieve ideal CPI of 1
- Now look at techniques to decrease CPI to less than 1
- To do this, must issue more than one instruction per cycle
- Multiple Issue Machines: Two flavors

1. Superscalar

- Issue varying numbers of instructions per clock (typically 1 to 8)
- Alpha tries to do 4; PowerPC up to 8
- May be scheduled by compiler or dynamically scheduled in hardware (scoreboard, Tomasulo)

2. VLIW: Very Long Instruction Word

- Very long instruction word
- Holds many instructions in fixed fields (maybe 16)
- Scheduled by the compiler -- know they can execute simultaneously
- Performance depends on success of compiler
- All dependencies resolved, so hardware doesn't have to check for them
- Makes instruction issue logic easy
- Fill in with NOPs if unable to find independent instructions
- So far, these machines not successful in the marketplace -- can't schedule enough concurrent instructions
- Joint HP/Intel agreement: 1998? Perhaps some brakthrough in compiler technology that will allow them to schedule more concurrent instructions

- **Superscalar DLX example**
- We will only look at static scheduling from the compiler (can also do dynamic scheduling)
- Assume two instructions can be issued per cycle

1. 1 floating point
2. 1 anything else (ld, store, br, integer ALU)

- Fetch 64 bits per clock cycle
- Integer instruction on left, floating point on right
- Can only issue 2nd instruction if 1st instruction issues\
- Need more ports for floating point register file: potentially do a FP load and FP operation on a single instruction pair

- Draw Fig. 4.26 in book:

Int Instr	IF	ID	EX	MEM	WB		
FP Instr	IF	ID	EX	MEM	WB		
Int Instr		IF	ID	EX	MEM	WB	
FP Instr		IF	ID	EX	MEM	WB	

- One problem: more expensive delays for loads on these machines!
- Now a single-cycle load delay actually delays three instructions from issuing
- Need to find three instructions to fill load delay slot
- **Recall: Loop unrolling example for avoiding branch delays**

```
for (i=1; i<=1000; i++)
        x[i] = x[i] + s;
```

- DLX code:

```
Loop: LD   F0, 0(R1)  ; get next vector element
      ADDD F4, F0, F2;  add scalar in F2
      SD  0(R1), F4;     store result
      SUBI R1, R1, 8;    decrement pointer 8 bytes (per double word)
      BNEZ  R1, Loop
```

- **Remember we unrolled the loop 4 times, got down to 14 cycles**
- **3.5 cycles per loop, 3 cycles do useful work**
- **How well can we do on scheduling this same unrolled loop on a superscalar machine?**
- Show overhead projection: Figure 4.27
- Now we unroll the loop five times
- 12 cycles for 5 loops, or 2.4 cycles per element
- **Limitations**
- Split FP/Integer instructions for simplicity in hardware
- Get CPI of 0.5 only for programs with :

1. Exactly 505 FP operations
2. No hazards

- For more instructions to issue at the same time, or more general mix of instructions
    - Greater difficulty of decode and issue
    - Even if 2-scalar: examine 2 opcodes, 6 register specifiers, decide if 1 or 2 instructions can issue based on dependencies
- **Loop Unrolling in VLIW: Figure 4.29**

- VLIW: can reduce hardware need to implement multi-isue machine
- Depend on compiler
- Use multiple, independent functional units
- Compiler decides which instructions to include, so hardware needn't decide
- **Example: Figure 4.29**
- Unrolled loop 7 times to minimize delays
- 2 memory references, 2 floating point operations, 1 integer/branch
- 7 results in 9 cycles, or 1.3 cycles per iteration
- Need more registers in VLIW

- **Limitations of Multi-Issue Machines**

1. **Inherent limitations of ILP**

- May have to unroll loops many times to get enough instructions to fill issue slots
- Especially for VLIW
- If units have long latency, may have to schedule many operations
- Need about as many operations as (pipeline depth * number functional units)

2. **Difficulties in Building Hardware**

- Duplicate functional units--not hard, but potentially expensive
- With multiple functional units, need increased number of ports to memory and register files

3. **Superscalar:**

- Complicated decoding logic, may have impact on clock rate and pipeline depth

4. **VLIW:**

- Increased code size (ambitiously unrolling loops, also including lots of NOPs)
    - About 1/2 of instructions typically empty
- Lock-step operations: stall in any functional unit stalls all instructions
- Binary compatibility: different generations will have different numbers of functional units

- **Hardware Support For Extracting More Parallelism**
- Compilers only succeed if branch behavior is fairly predictable at compile time
- Hardware techniques:

1. **Conditional or Predicated Instructions**

- Extension of instruction seet
- Conditional instruction: instruction refers to condition, which is evaluated as part of instruction execution
- If true, execute normally
- If false, turn into no-op
- Most common example: conditional move: moves value from one register to another if condition is true
- Can eliminate branch in simple code sequences
- Example: if (A=0) {B=C;}

  DLX code: if R1, R2, R3 hold A, B, C

```
      BNEZ R1, TARGET
      MOV R2, R3
TARGET: ....
```

  With conditional MOV instruction:

```
      CMOVZ R2, R3, R1
```

  Can also move instructions before the branch

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

  ** Moves place in pipeline where dependence must be resolved from front of pipeline where branch is evaluated to end of pipeline, where register write occurs

  Can use it to move instructions across the branch

2. **Hardware Speculation**

   Combination of

- **dynamic branch prediction,**
- **speculation to allow execution of instructions before control dependences are resolved,**
- **dynamic scheduling (e.g., Tomasulo's algorithm) to deal with scheduling of different combinations of bsic blocks**

- Extension of Tomasulo's algorithm
- Separate bypassing of results among instructions from actual completion of instrcution
- Allow an instruction to execute and bypass its result to other isntructions
- Without allowing instructions to change state -- perform updates that cannot be undone--until we know the instruction is no longer speculative
- Instruction commit: update register file or memory when no longer speculative
- **Key idea: allow instructions to execute out of order, but force them to commit in order**
- Need additional set of hardware buffers to hold results of isntructions that have completed execution but not yet committed
- Reorder buffer (addition to reservation stations, but hold speculative results)
- Phases of execution:

1. Issue: issue instruction if empty reservation station and empty slot in reorder buffer

   if operands in reservation stations or in reorder buffer, send to reservation station
2. Execute: if waiting for operands, monitor CDB

   check for RAW hazards

   When both operands available at reservation station, execute
3. Write result: when result available, write on CDB and into re-order buffer and into any reservation stations waiting for the result
4. Commit: instructions move to head of reorder buffer queue

   Branch with incorrect prediction: speculation was wrong, so flush reorder buffer and restart execution at correct branch successor

   Otherwise, update the register with result or perform memory write and remove instruction from reorder buffer

- **From Paper on Instruction-Level Parallelism**
- **ILP: objective is execution in parallel of lowest level machine operations (loads, stores, integer, FP)**
- Distinct from parallel programs: require major algorithm and code changes
- Example used throughout paper: processor that can perform two integer operations, two load/stores and one branch/compare every cycle
- Simple example ignores FP instructions
- Requires multiported register file or several multiported register banks
- Key issue is making good use of all the potential parallelism

- Look at code:
    - Basic block: contiguous code without branches
    - Serial execution: each instruction completes before next is begun
    - Basic block ILP: use instruction parallelism with a basic block
    - As soon as operands are available, let instructions execute
    - Limited by small amount of parallelism within basic block: instructions fairly dependent
    - Can make a graph of dependencies between instructions
    - Longest path is critical path
- Global ILP
    - Basic blocks separated mostly by conditional branches
    - Only when condition is known can you know which is the next basic block
    - How to get parallelism between blocks?
    - One way is to identify "control parallelism": certain instructions will be executed regardless of the outcome of the branch
    - Another approach: "speculative execution": predict branch outcome
- Potential ILP
    - Within basic blocks: speedup limited from 2X to 3.5X
    - Across basic blocks: 5X to 20X
- ILP Architectures:
    - Sequential architectures: program does not convey information about instruction dependencies; hardware detects hazards and schedules (example: superscalar)
    - Dependence architectures: program explicitly indicates dependencies; hardware uses this info to schedule instructions (example: dataflow)
    - Independences architectures: compiler determines all dependencies and does all scheduling (example: VLIW)
- Superscalar
    - We have already discussed these
    - Mention Tomasulo's algorithm for resolving hazards, doing HW scheduling
    - Turns out to have been the most successful approach because lack of inherent parallelism limits the other approaches
    - Typical ILP available allows 2X to 3X speedup
- Dataflow
    - These processors were a very "hot" topic at this time
    - Monitor operand availability

- When operands are available, fetch instruction that uses them
    - Wait for functional unit to become available
    - Execute
    - Tries to look far down the execution path to try to utilize functional units
    - Did not do speculation
    - Not commercially successful: not able to find adequate parallelism
- VLIW
    - Compiler detects all dependencies, does all scheduling
    - Can use speculative execution
    - Again, not commercially successful
    - Insufficient parallelism/independence to keep the VLIW full of useful instructions

- Compiler Techniques
- **Software Pipelining**
    - Similar analysis to loop unrolling in some ways
    - Constructs a new loop that contains instructions from different iterations of the loop
    - Then all can be done completely in parallel
    - Eliminates the problem we had in loop unrolling when we had to wait for delays from the load to complete the add and for the add before completing store
    - Have some code at beginning and the end to take care of bookkeeping--any instructions thattt don't make it into the software pipelined loop
    - Software counterpart to what Tomasulo's algorithm does in hardware
    - Figures 4.30, 4.31
    - Example on p. 294
- Differences from looop unrolling:
- Loop unrolling reduces the overhead associated with running the loop
    - Our example: unrolled the loop four times, so only paid overhead once for every four iterations of the loop
- Software pipelining keeps the loop running at top speed for much longer
    - Does include the branch overhead instructions
    - So, techniques are actually complimentary

- **Trace Scheduling**
  - Trace selection: compiler chooses a likely sequence of basic blocks, perhaps based on profiling
  - Then schedule/compact the trace: find inepependent instructions
  - So, scheduling across basic blocks and not just iterations of a loop
  - Allows code to move across branches
  - Some code movement will be speculative, based on branch predictions
  - Not allowed to generate exceptions that would not otherwise happen
  - Also can't change state with speculative instructions