Instruction-Level Parallelism Basics

Instruction-Level Parallelism

- Basics
- Compiler techniques
- Loop-level parallelism
- Branch prediction
- Multiple issue and static scheduling
- Multiple issue and dynamic scheduling
- Advanced techniques for instruction delivery

Introduction

- Pipelining become a universal technique in 1985.
  - Overlaps execution of instructions
  - Exploits “instruction-level parallelism”
- Beyond this, there are two main approaches.
  - Hardware-based dynamic approaches
    - Used in server and desktop processors
    - Not used as extensively in multicomputers
  - Compiler-based static approaches
    - Not as successful outside of scientific applications

Instruction-Level Parallelism

- When exploiting instruction-level parallelism, goal is to maximize CPI.
  - Pipeline CPI =
    - Ideal pipeline CPI +
    - Structural stalls +
    - Data hazard stalls +
    - Control stalls
- Parallelism with basic block is limited.
  - Typical size of basic block = three to six instructions
  - Must optimize across branches

Data Dependence

- Loop-level parallelism
  - Unroll loop statically or dynamically.
  - Use SIMD (vector processors and GPUs).
- Challenges:
  - Data dependency
    - Instruction *j* is data dependent on instruction *i* if
      - Instruction *i* produces a result that may be used by instruction *j.*
      - Instruction *j* is data dependent on instruction *k* and instruction *k* is data dependent on instruction *i.*
- Dependent instructions cannot be executed simultaneously.

Data Dependence (cont.)

- Dependencies are a property of programs.
- Pipeline organization determines if dependence is detected and if it causes a stall.
- Data dependence conveys:
  - Possibility of a hazard
  - Order in which results must be calculated
  - Upper bound on exploitable instruction-level parallelism
- Dependencies that flow through memory locations are difficult to detect.

Name Dependence

- Two instructions use the same name but no flow of information.
  - Not a true data dependence, *but is a problem when reordering instructions.*
  - *Antidependence*: Instruction *j* writes a register or memory location that instruction *i* reads.
    - Initial ordering (*i* before *j*) must be preserved.
  - *Output dependence*: Instruction *i* and instruction *j* write the same register or memory location.
    - Ordering must be preserved.
- To resolve, use renaming techniques.

Other Factors

- Data hazards
  - Read after write (RAW)
  - Write after write (WAW)
  - Write after read (WAR)
- Control dependence
  - Ordering of instruction *i* with respect to a branch instruction
    - Instruction control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch.
    - An instruction not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch.

Examples

- Example 1:

```
    DADDU     R1,R2,R3
    BEQZ      R4,L
    DSUBU     R1,R1,R6
L:  …
    OR        R7,R1,R8
```

- OR instruction dependent on DADDU and DSUBU.

- Example 2:

```
    DADDU     R1,R2,R3
    BEQZ      R12,skip
    DSUBU     R4,R5,R6
    DADDU     R5,R4,R9
skip:
    OR        R7,R8,R9
```

- Assume R4 isn't used after skip.
- Possible to move DSUBU before the branch.

ENGINEERING@SYRACUSE

Compiler Techniques

Compiler Techniques for Exposing ILP

- Pipeline scheduling
  - Separate dependent instruction from the source instruction by the pipeline latency of the source instruction.
- Example:

```
for (i=999; i>=0; i=i-1)
  x[i] = x[i] + s;
```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Pipeline Stalls

```
Loop:  L.D    F0,0(R1)
       stall
       ADD.D  F4,F0,F2
       stall
       stall
       S.D    F4,0(R1)
       DADDUI R1,R1,#-8
       stall (assume integer load latency is 1)
       BNE    R1,R2,Loop
```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Pipeline Scheduling

Scheduled code:

```
Loop:  L.D    F0,0(R1)
       DADDUI R1,R1,#-8
       ADD.D  F4,F0,F2
       stall
       stall
       S.D    F4,8(R1)
       BNE    R1,R2,Loop
```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Loop Unrolling

- Unroll by a factor of four (assume # elements is divisible by four).
- Eliminate unnecessary instructions.

```
Loop:   L.D     F0,0(R1)
        ADD.D   F4,F0,F2
        S.D     F4,0(R1);       drop DADDUI & BNE
        L.D     F6,-8(R1)
        ADD.D   F8,F6,F2
        S.D     F8,-8(R1);      drop DADDUI & BNE
        L.D     F10,-16(R1)
        ADD.D   F12,F10,F2
        S.D     F12,-16(R1);    drop DADDUI & BNE
        L.D     F14,-24(R1)
        ADD.D   F16,F14,F2
        S.D     F16,-24(R1)
        DADDUI R1,R1,#-32
        BNE     R1,R2,Loop
```

Note: number of live registers vs. original loop

Loop Unrolling/Pipeline Scheduling

- Pipeline schedule the unrolled loop:

```
Loop:   L.D     F0,0(R1)
        L.D     F6,-8(R1)
        L.D     F10,-16(R1)
        L.D     F14,-24(R1)
        ADD.D   F4,F0,F2
        ADD.D   F8,F6,F2
        ADD.D   F12,F10,F2
        ADD.D   F16,F14,F2
        S.D     F4,0(R1)
        S.D     F8,-8(R1)
        DADDUI  R1,R1,#-32
        S.D     F12,16(R1)
        S.D     F16,8(R1)
        BNE     R1,R2,Loop
```

Strip Mining

- Unknown number of loop iterations
  - Number of iterations = *n.*
  - Goal: Make *k* copies of the loop body.
  - Generate pair of loops:
    - First executes *n* mod *k* times.
    - Second executes *n* / *k* times.
    - “Strip mining.”

ENGINEERING@SYRACUSE

Loop-Level Parallelism

Loop-Level Parallelism

- Focuses on determining whether data accessed in later iterations are dependent on data values produced in earlier iterations
  - Loop-carried dependence

- Example 1:

```
for (i=999; i>=0; i=i-1)
    x[i] = x[i] + s;
```

- No loop-carried dependence

Loop-Level Parallelism (cont.)

- Example 2:

```
for (i=0; i<100; i=i+1) {
        A[i+1] = A[i] + C[i];   /* S1 */
        B[i+1] = B[i] + A[i+1]; /* S2 */
      }
```

- S1 and S2 use values computed by S1 in previous iteration.
- S2 uses value computed by S1 in same iteration.

Loop-Level Parallelism (cont.)

- Example 3:

```
for (i=0; i<100; i=i+1) {
   A[i] = A[i] + B[i];   /* S1 */
   B[i+1] = C[i] + D[i]; /* S2 */
      }
```

- S1 uses value computed by S2 in previous iteration, but dependence is not circular so loop is parallel.

- Transform to:

```
A[0] = A[0] + B[0];
for (i=0; i<99; i=i+1) {
      B[i+1] = C[i] + D[i];
      A[i+1] = A[i+1] + B[i+1];
}
B[100] = C[99] + D[99];
```

Loop-Level Parallelism (cont.)

- Example 4:

```
for (i=0;i<100;i=i+1)  {
  A[i] = B[i] + C[i];
  D[i] = A[i] * E[i];
}
```

- A[i] can be computed into a register in S1, which can then be read in S2.

- Example 5:

```
for (i=1;i<100;i=i+1)  {
   Y[i] = Y[i-1] + Y[i];
}
```

- Recurring loop-carried dependences.

Finding Dependencies

- Assume indices are affine:

  `V[a*i + b] (i is loop index)`

- Assume:
  - Store to **`V[a*i + b]`**, then…
  - Load from **`V[c*i + d].`**
  - `i` runs from `m` to `n.`
  - Dependence exists if:
    - Given `j, k` such that `m ≤ j ≤ n, m ≤ k ≤ n`…
    - Store to `[a*j + b],`
    - Load from `[a*k + d],`
    - and `[a*j + b] = [c*k + d].`

Finding Dependencies

- Generally cannot determine at compile time
- Test for absence of a dependence:
  - GCD test:
    - If a dependency exists, GCD(*c*,*a*) must evenly divide (*d*-*b*).

- Example 1:

```
for (i=0; i<100; i=i+1) {
      X[2*i+3] = X[2*i] * 5.0;
   }
```

- (a,b,c,d)=(2,3,2,0)
- GCD (a,c) = 2, d-b = -3

2 doesn't divide 3 ➔
No dependence possible.

Finding Dependencies (cont.)

- Example 2:

```
for (i=0; i<100; i=i+1) {
   Y[i] = X[i] / c; /* S1 */
   X[i] = X[i] + c; /* S2 */
   Z[i] = Y[i] + c; /* S3 */
   Y[i] = c - Y[i]; /* S4 */
}
```

- S1 → S3; S1 → S4 on Y:
  ➔ True dependency, not loop-carried.
- S1 → S2 on X; S3 → S4 on Y:
  ➔ Antidependency
- S1 → S4 on Y:
  ➔ Output dependecy

- Watch for antidependencies and output dependencies.

Finding Dependencies (cont.)

- Example 2:

```
for (i=0; i<100; i=i+1) {
  T[i] = X[i] / c; /* Y renamed to T to remove OD
  X1[i] = X[i] + c;/* X renamed to X1 to remove AD
  Z[i] = T[i] + c; /* Y renamed to T to remove AD
  Y[i] = c - T[i]; /*
}
```

- Watch for antidependencies and output dependencies.
- Variable x has been renamed x1.

Reductions

- Reduction operation:

```
for (i=9999; i>=0; i=i-1)
  sum = sum + x[i] * y[i];
```

- Transform to…

```
for (i=9999; i>=0; i=i-1)
  sum [i] = x[i] * y[i];
for (i=9999; i>=0; i=i-1)
  finalsum = finalsum + sum[i];
```

- Do on p processors:

```
for (i=999; i>=0; i=i-1)
   finalsum[p] = finalsum[p] + sum[i+1000*p];
```

Note: Assumes associativity!

ENGINEERING@SYRACUSE

Dynamic Scheduling

Dynamic Scheduling

- Rearrange order of instructions to reduce stalls while maintaining data flow.
- Advantages:
  - Compiler doesn't need to have knowledge of microarchitecture.
  - Handles cases where dependencies are unknown at compile time.
- Disadvantages:
  - Substantial increase in hardware complexity
  - Complicates exceptions

Dynamic Scheduling (cont.)

- Dynamic scheduling implies:
  - Out-of-order execution
  - Out-of-order completion
- Creates the possibility for WAR and WAW hazards
- Tomasulo's approach
  - Tracks when operands are available
  - Introduces register renaming in hardware
    - Minimizes WAW and WAR hazards

Register Renaming

- Example:

```
DIV.D     F0,F2,F4
ADD.D     F6,F0,F8
S.D       F6,0(R1)
SUB.D     F8,F10,F14
MUL.D     F6,F10,F8
```

+ name dependence with F6

Register Renaming

- Example:

```
DIV.D     F0,F2,F4
ADD.D     S,F0,F8
S.D       S,0(R1)
SUB.D     T,F10,F14
MUL.D     F6,F10,T
```

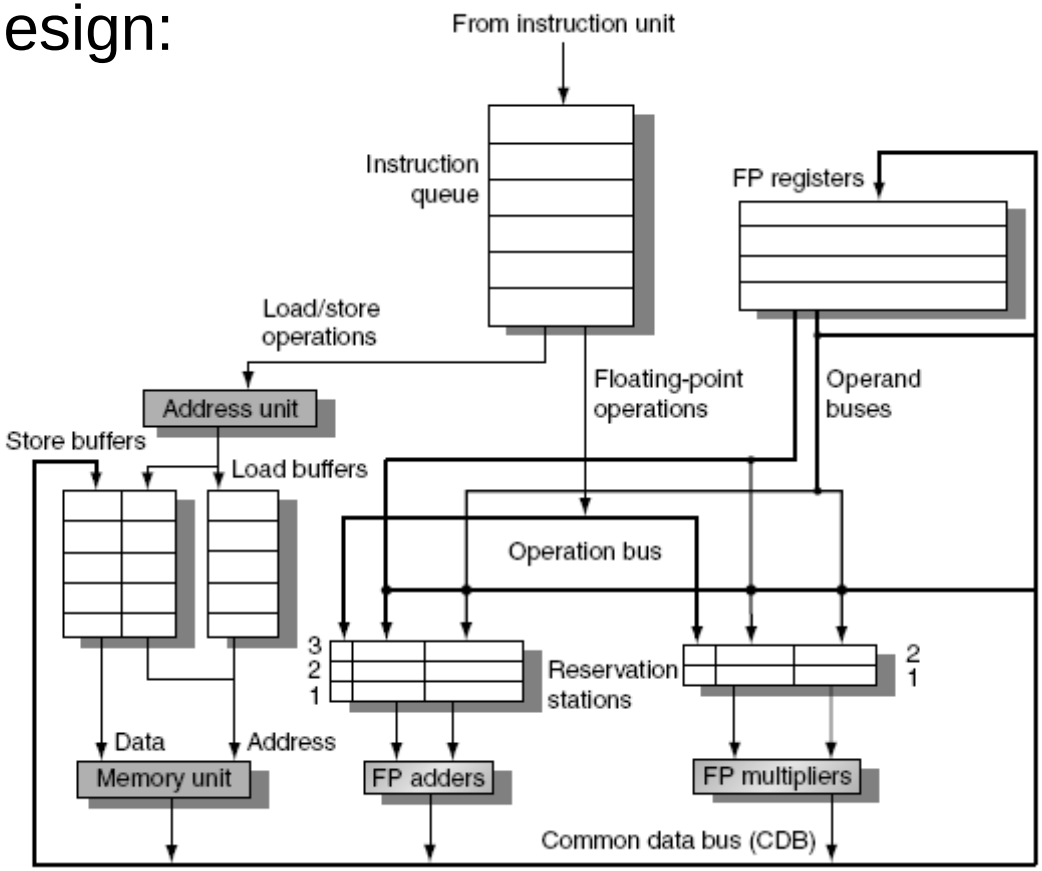
- Now only RAW hazards remain, which can be strictly ordered.

Register Renaming (cont.)

- Register renaming is provided by reservation stations (RS).
  - Contains:
  - The instruction
  - Buffered operand values (when available)
  - RS number of instruction providing the operand values
- RS fetches and buffers an operand as soon as it becomes available.
- Pending instructions designate the RS to which they will send their output.
  - Result values broadcast on a result bus, called the common data bus (CDB).
- Only the last output updates the register file.
- As instructions are issued, the register specifiers are renamed with the reservation station.
- May be more reservation stations than registers.

Tomasulo's Algorithm

- Load and store buffers
  - Contain data and addresses, act like reservation stations
- Top-level design:



Tomasulo's Algorithm (cont.)

- Three steps:
  - Issue.
    - Get next instruction from FIFO queue.
    - If available RS, issue the instruction to the RS with operand values if available.
    - If operand values not available, stall the instruction.
  - Execute.
    - When operand becomes available, store it in any reservation stations waiting for it.
    - When all operands are ready, issue the instruction.
    - Loads and store maintained in program order through effective address.
    - No instruction allowed to initiate execution until all branches that proceed it in program order have completed.
  - Write result.
    - Write result on CDB into reservation stations and store buffers.
      - (Stores must wait until address and value are received.)

Example

Instruction		Instruction status		
		Issue	Execute	Write Result
L.D	F6,32(R2)	√	√	√
L.D	F2,44(R3)	√	√	
MUL.D	F0,F2,F4	√		
SUB.D	F8,F2,F6	√		
DIV.D	F10,F0,F6	√		
ADD.D	F6,F8,F2	√		

	Reservation stations						
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	No						
Load2	Yes	Load					44 + Regs[R3]
Add1	Yes	SUB		Mem[32 + Regs[R2]]	Load2		
Add2	Yes	ADD			Add1	Load2	
Add3	No						
Mult1	Yes	MUL		Regs[F4]	Load2		
Mult2	Yes	DIV		Mem[32 + Regs[R2]]	Mult1		

	Register status								
Field	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi	Mult1	Load2		Add2	Add1	Mult2			

Speculation

- Predict branch and continue issuing.
  - Don’t commit until branch outcome determined.
- Load speculation.
  - Avoid load and cache miss delay.
    - Predict the effective address.
    - Predict loaded value.
    - Load before completing outstanding stores.
    - Bypass stored values to load unit.
  - Don’t commit load until speculation cleared.

Hardware-Based Speculation

- Execute instructions along predicted execution paths, but only commit the results if prediction was correct.
- Instruction commit: allowing an instruction to update the register file when instruction is no longer speculative.
- Need an additional piece of hardware to prevent any irrevocable action until an instruction commits.
  - I.e., updating state or taking an execution

Reorder Buffer

- Reorder buffer: holds the result of instruction between completion and commit

- Four fields:
  - Instruction type: branch/store/register
  - Destination field: register number
  - Value field: output value
  - Ready field: completed execution?

- Modify reservation stations:
  - Operand source is now reorder buffer instead of functional unit.

Reorder Buffer (cont.)

- Register values and memory values are not written until an instruction commits.
- On misprediction:
  - Speculated entries in ROB are cleared.
- Exceptions:
  - Not recognized until it is ready to commit

ENGINEERING@SYRACUSE

Multiple Issue and Static Scheduling

Multiple Issue

- Static multiple issue
  - Compiler groups instructions to be issued together.
  - Packages them into “issue slots.”
  - Compiler detects and avoids hazards.
- 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.

Static Multiple Issue

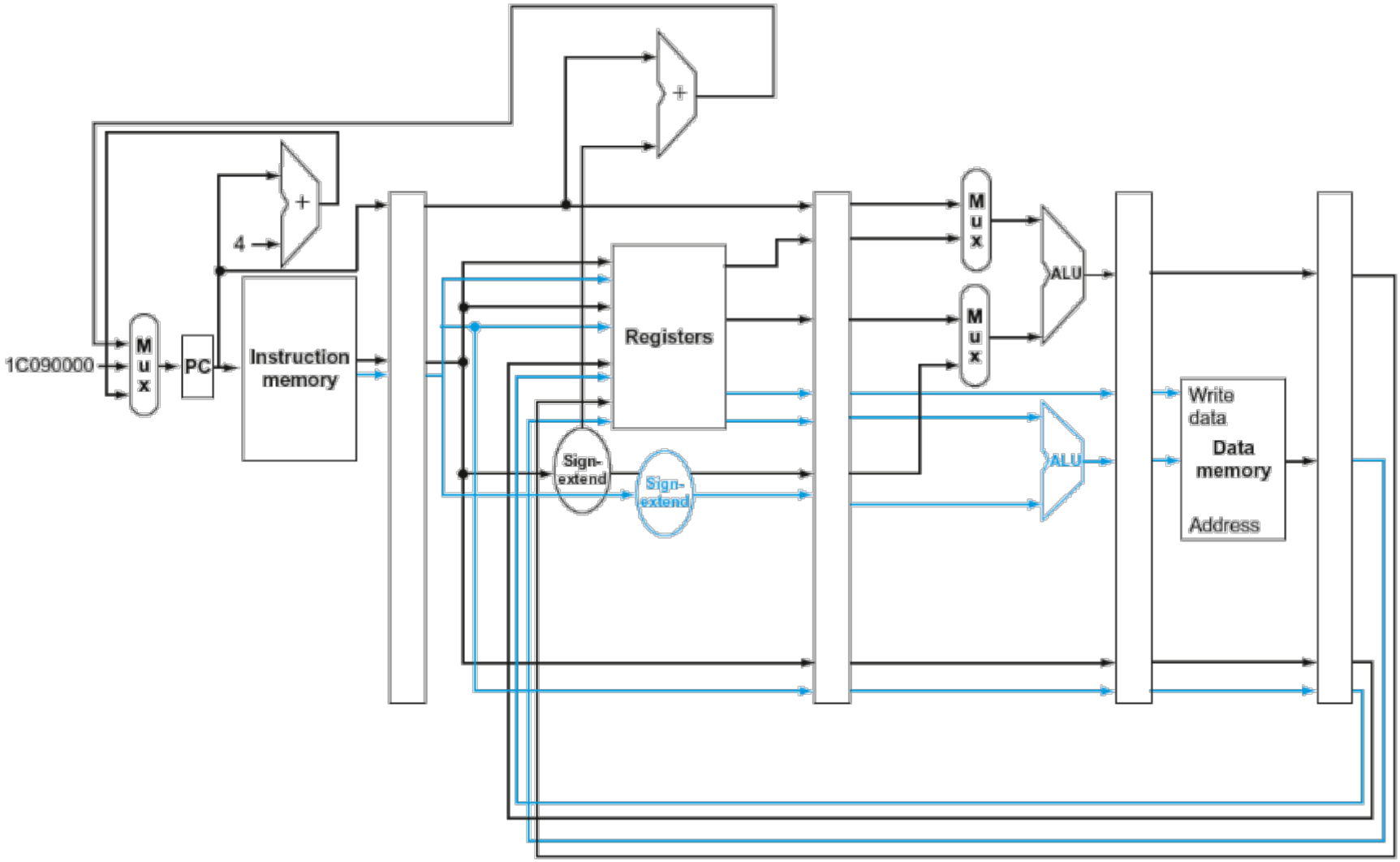
- Compiler groups instructions into “issue packets.”
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required
- Think of an issue packet as a very long instruction.
  - Specifies multiple concurrent operations
  - $\Rightarrow$ Very long instruction word (VLIW)

Av8 with Static Dual Issue

- Two-issue packets
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store.
    - Pad an unused instruction with nop.

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

Av8 with Static Dual Issue (cont.)



Multiple Issue

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the ARM Coretex A8
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

VLIW Processors

- Package multiple operations into one instruction
- Example VLIW processor:
  - One integer instruction (or branch)
  - Two independent floating-point operations
  - Two independent memory references
- Must be enough parallelism in code to fill the available slots

VLIW Processors (cont.)

Disadvantages:

- Statically finding parallelism
- Code size
- No hazard detection hardware
- Binary code compatibility

ENGINEERING@SYRACUSE

Multiple Issue and Dynamic Scheduling

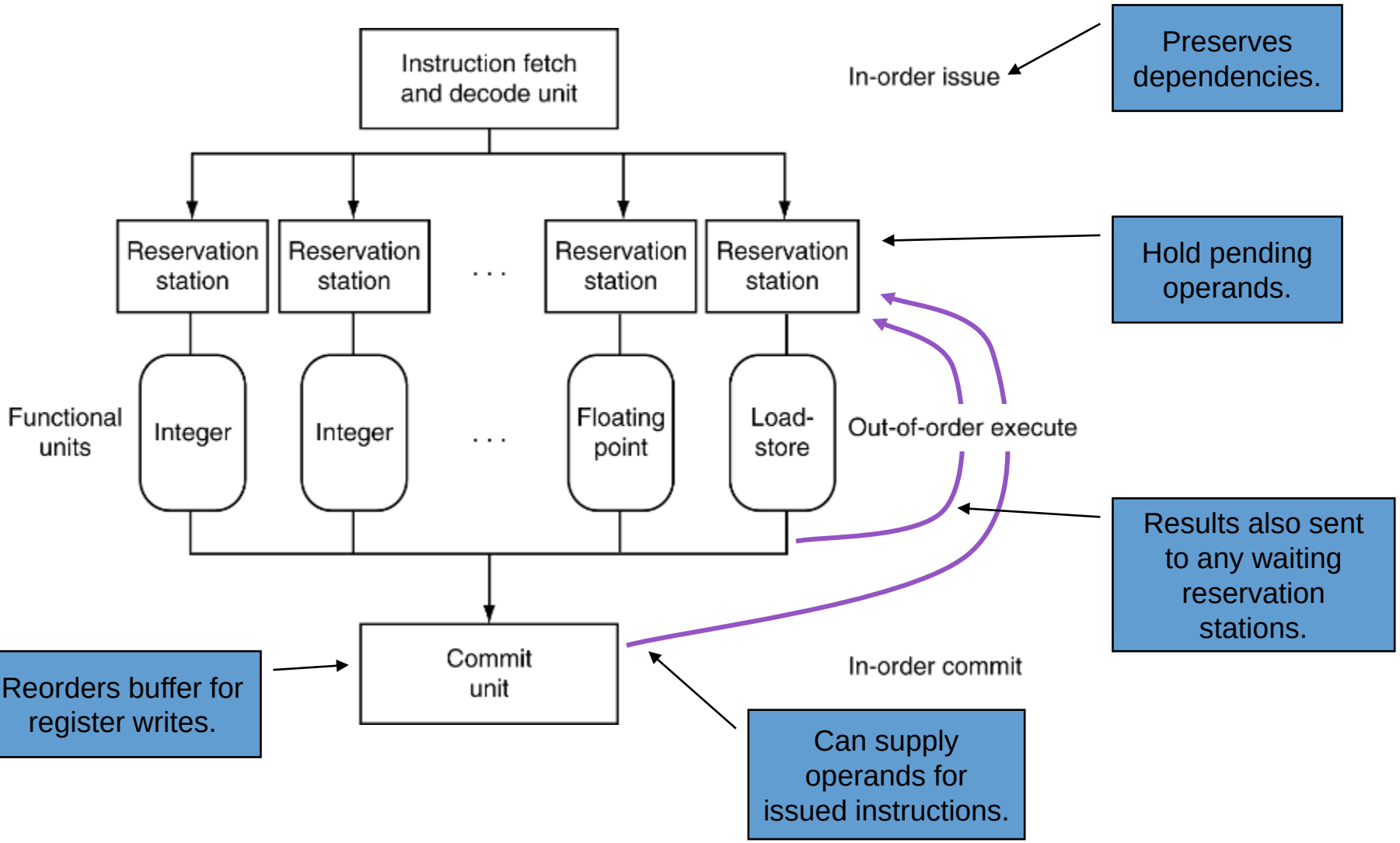
Why Do Dynamic Scheduling?

- 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.

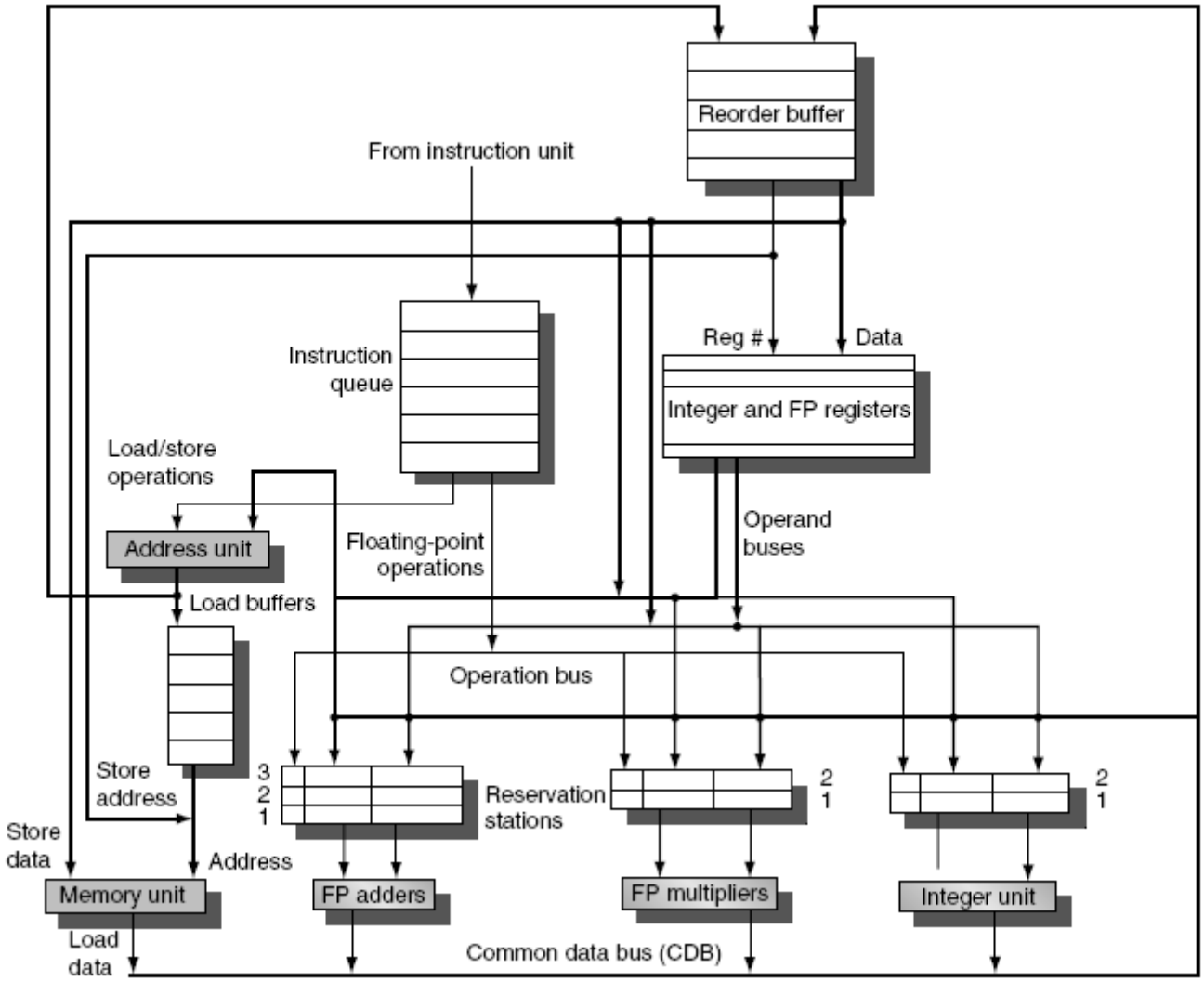
Dynamic Scheduling

- Modern microarchitectures:
  - Dynamic scheduling + multiple issue + speculation
- Two approaches:
  - Assign reservation stations and update pipeline control table in half clock cycles.
    - Only supports two instructions/clock
  - Design logic to handle any possible dependencies between the instructions.
  - Hybrid approaches.
- Issue logic can become bottleneck.

Dynamically Scheduled CPU



Overview of Design



Multiple Issue

- Limit the number of instructions of a given class that can be issued in a “bundle.”
  - I.e., on FP, one integer, one load, one store
- Examine all the dependencies among the instructions in the bundle.
- If dependencies exist in bundle, encode them in reservation stations.
- Also need multiple completion/commit.

Example

```
Loop: LD    R2,0(R1)   ;R2=array element
      DADDIU     R2,R2,#1   ;increment R2
      SD    R2,0(R1)   ;store result
      DADDIU R1,R1,#8  ;increment pointer
      BNE   R2,R3,LOOP ;branch if not last element
```

Example (No Speculation)

Iteration number	Instructions		Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD	R2,0(R1)	1	2	3	4	First issue
1	DADDIU	R2,R2,#1	1	5		6	Wait for LW
1	SD	R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU	R1,R1,#8	2	3		4	Execute directly
1	BNE	R2,R3,LOOP	3	7			Wait for DADDIU
2	LD	R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU	R2,R2,#1	4	11		12	Wait for LW
2	SD	R2,0(R1)	5	9	13		Wait for DADDIU
2	DADDIU	R1,R1,#8	5	8		9	Wait for BNE
2	BNE	R2,R3,LOOP	6	13			Wait for DADDIU
3	LD	R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU	R2,R2,#1	7	17		18	Wait for LW
3	SD	R2,0(R1)	8	15	19		Wait for DADDIU
3	DADDIU	R1,R1,#8	8	14		15	Wait for BNE
3	BNE	R2,R3,LOOP	9	19			Wait for DADDIU

Example

Iteration number	Instructions		Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	LD	R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU	R2,R2,#1	1	5		6	7	Wait for LW
1	SD	R2,0(R1)	2	3			7	Wait for DADDIU
1	DADDIU	R1,R1,#8	2	3		4	8	Commit in order
1	BNE	R2,R3,LOOP	3	7			8	Wait for DADDIU
2	LD	R2,0(R1)	4	5	6	7	9	No execute delay
2	DADDIU	R2,R2,#1	4	8		9	10	Wait for LW
2	SD	R2,0(R1)	5	6			10	Wait for DADDIU
2	DADDIU	R1,R1,#8	5	6		7	11	Commit in order
2	BNE	R2,R3,LOOP	6	10			11	Wait for DADDIU
3	LD	R2,0(R1)	7	8	9	10	12	Earliest possible
3	DADDIU	R2,R2,#1	7	11		12	13	Wait for LW
3	SD	R2,0(R1)	8	9			13	Wait for DADDIU
3	DADDIU	R1,R1,#8	8	9		10	14	Executes earlier
3	BNE	R2,R3,LOOP	9	13			14	Wait for DADDIU

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.

Power Efficiency

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

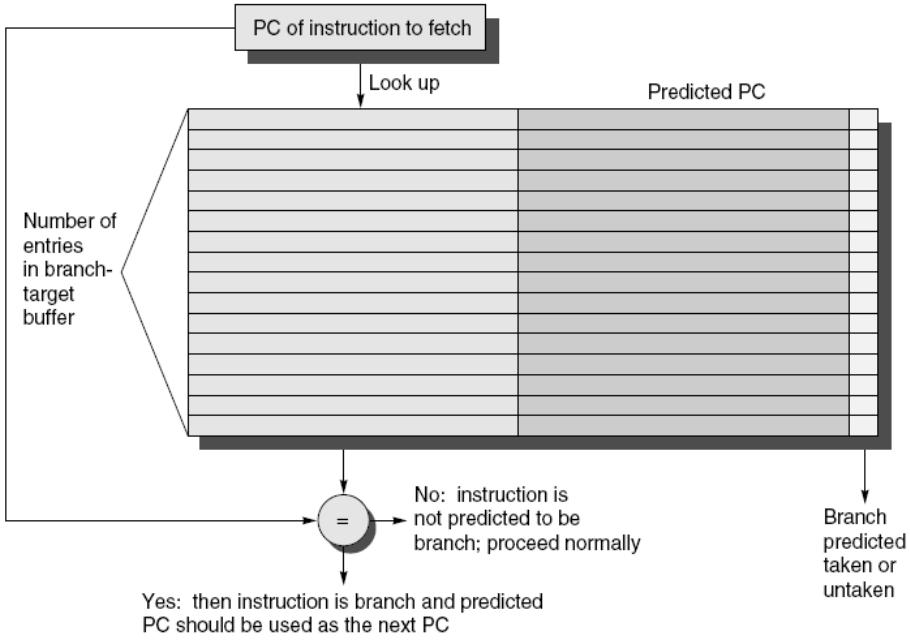
Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-Order/ Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2000MHz	22	3	Yes	1	75W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W
Core	2006	2930MHz	14	4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

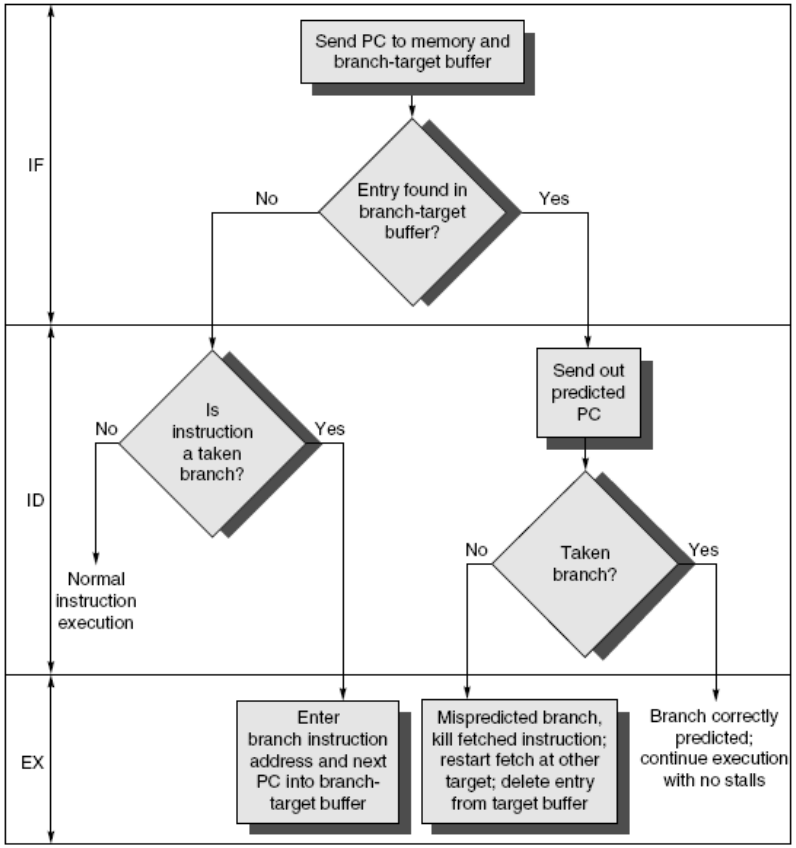
ENGINEERING@SYRACUSE

Advanced Techniques for Instruction Delivery

Branch-Target Buffer

- Need high instruction bandwidth!
  - Branch-target buffers
    - Next PC prediction buffer, indexed by current PC





Branch Folding

- Optimization:
  - Larger branch-target buffer.
  - Add target instruction into buffer to deal with longer decoding time required by larger buffer.
  - “Branch folding.”

Return Address Predictor

- Most unconditional branches come from function returns.
- The same procedure can be called from multiple sites.
  - Causes the buffer to potentially forget about the return address from previous calls
- Create return address buffer organized as a stack.

Integrated Instruction Fetch Unit

- Design monolithic unit that performs:
  - Branch prediction
  - Instruction prefetch
    - Fetch ahead.
  - Instruction memory access and buffering
    - Deal with crossing cache lines

Register Renaming

- Register renaming vs. reorder buffers
  - Instead of virtual registers from reservation stations and reorder buffer, create a single register pool.
    - Contains visible registers and virtual registers
  - Use hardware-based map to rename registers during issue.
  - WAW and WAR hazards are avoided.
  - Speculation recovery occurs by copying during commit.
  - Still need a ROB-like queue to update table in order.
  - Simplifies commit:
    - Record that mapping between architectural register and physical register is no longer speculative.
    - Free up physical register used to hold older value.
    - In other words: Swap physical registers on commit.
  - Physical register deallocation is more difficult.

Integrated Issue and Renaming

- Combining instruction issue with register renaming:
  - Issue logic prereserves enough physical registers for the bundle (fixed number?).
  - Issue logic finds dependencies within bundle, and maps registers as necessary.
  - Issue logic finds dependencies between current bundle and already in-flight bundles, and maps registers as necessary.

How Much?

- How much to speculate
  - Misspeculation degrades performance and power relative to no speculation.
    - May cause additional misses (cache, TLB)
  - Prevent speculative code from causing higher costing misses (e.g., L2).

- Speculating through multiple branches
  - Complicates speculation recovery.
  - No processor can resolve multiple branches per cycle.

Energy Efficiency

- Speculation and energy efficiency
  - Note: Speculation is energy efficient only when it significantly improves performance.

- Value prediction
  - Uses:
    - Loads that load from a constant pool
    - Instruction that produces a value from a small set of values
  - Not been incorporated into modern processors.
  - Similar idea—*address aliasing prediction*—is used on some processors.

ENGINEERING@SYRACUSE

Conclusions

In Conclusion

- ILP
  - Maximize CPI, then use IPC.
  - Interiteration dependencies.
- Loops
  - Unrolling, software pipelining
- Dynamic scheduling
  - Out-of-order execution, speculation
- Multiple issue and static scheduling
  - Compile-time, fixed number of operations/instruction
  - VLIW
- Multiple issue and dynamic scheduling
  - Runtime, variable number of operations/instruction
  - Superscalar

ENGINEERING@SYRACUSE