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
  - Need to separate executing the instruction to pass data to other instructions from completing (performing operations that can not be undone)

Reorder Buffer

- Reorder buffer – holds the result of instruction between completion and commit (and supply them to any instruction who needs them just like the RS in Tomasulo's)
- Four fields:
  - Instruction type: branch/store/register
  - Destination field: register number or memory address
  - Value field: output value
  - Ready field: completed execution?
- Modify reservation stations:
  - Operand source is now reorder buffer instead of functional unit (results are tagged with ROB entry #)

Reorder Buffer

- 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
- 4 stages
  - Issue
  - Execute
  - Write Result
  - Commit

Reorder Buffer

- Issue
  - If empty RS and ROB entry → Issue; else stall
  - Send operands to RS if available in registers or ROB
  - The number of the ROB entry allocated to instruction is sent to RS to tag the results with
  - If operands are not available yet, the ROB entry is sent to the RS to wait for results on the CDB

Reorder Buffer

- Execute
  - If one or more operands are not available, monitor the CDB.
  - When the result is broadcast on the CDB (we know that from the ROB entry tag) copy it
  - When all operands are ready, start execution
- Write Result
  - When execution is completed, broadcast the result on the CDB tagged with ROB entry #
  - Results are copied to ROB entry and all waiting RS
- Execute out of order, commit in order.

Reorder buffer

- When an instruction reaches the head of the ROB and the result is ready in the buffer,
  - If ALU op write it to the register file and remove instruction from ROB
  - If the instruction is a store, write it to the memory and remove the instruction from the ROB
  - If the instruction is a branch, if prediction is correct, remove it from the ROB. If misprediction flush the ROB and start from the correct successor.

Multiple Issue and Static Scheduling

- To achieve CPI < 1, need to complete multiple instructions per clock
- Solutions:
  - Statically scheduled superscalar processors
  - VLIW (very long instruction word) processors
  - dynamically scheduled superscalar processors

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 the code to fill the available slots

VLIW Processors

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

VLIW Example

Source instruction	Instruction using result	Latency
FP ALU OP	FP ALU OP	3
FP ALU OP	Store double	2
Load double	FP ALU OP	1
Load Double	Store double	0

```
Loop: L.D      F0,0(R1)
      ADD.D    F4,F0,F2
      S.D      0(R1),F4
      DADDUI   R1,R1,#-8
      BNE  R   1,R2,Loop
```

For (I=1000;I>0;I++)

x[I]=x[I]+s;

VLIW Example

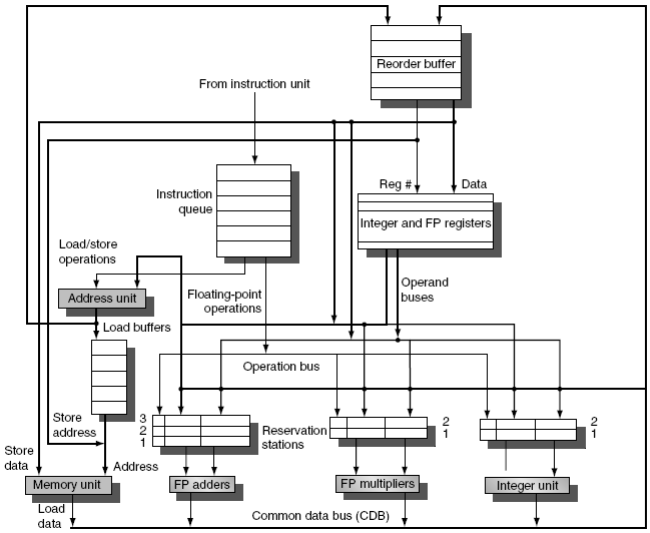
- Assume that w can schedule 2 memory operations, 2 FP operations, and one integer or branch

Memory reference 1	*Memory reference 2*	*FP operation 1*	*FP op. 2*	*Int. op/ branch*	*Clock*
LD F0,0(R1)	LD F6,-8(R1)				1
LD F10,-16(R1)	LD F14,-24(R1)				2
LD F18,-32(R1)	LD F22,-40(R1)	ADDD F4,F0,F2	ADDD F8,F6,F2		3
LD F26,-48(R1)		ADDD F12,F10,F2	ADDD F16,F14,F2		4
		ADDD F20,F18,F2	ADDD F24,F22,F2		5
SD 0(R1),F4	SD -8(R1),F8	ADDD F28,F26,F2			6
SD -16(R1),F12	SD -24(R1),F16			DADD R1,R1,#-56	7
SD 24(R1),F20	SD 16(R1),F24				8
SD 8(R1),F28				BNEZ R1,LOOP	9

Dynamic Scheduling, Multiple Issue, and Speculation

- Modern microarchitectures:
  - Dynamic scheduling + multiple issue + speculation
- Two approaches:
  - Assign reservation stations and update pipeline control table in half clock cycles
    - Only supports 2 instructions/clock
  - Design logic to handle any possible dependencies between the instructions
  - Hybrid approaches
- Issue logic can become bottleneck

Overview of Design



Multiple Issue

- Limit the number of instructions of a given class that can be issued in a “bundle”
  - I.e. one 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)

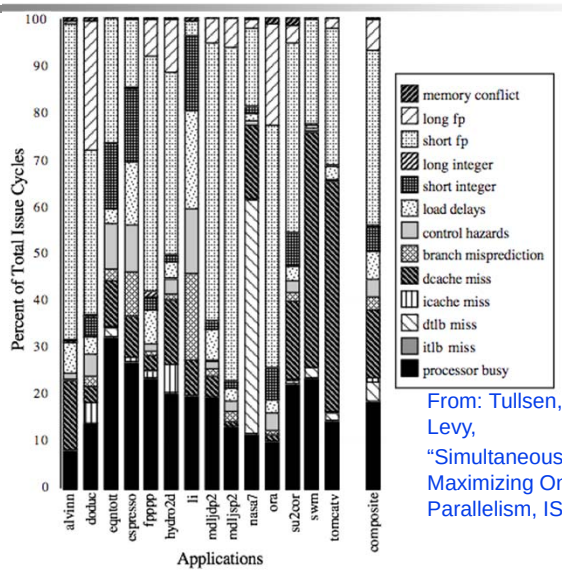
Iteration number	Instructions		Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD	R2,0(R1)	1	2	3	4	First issue
1	DADDIU	R2,R2,#1	1	5		6	Wait for LW
1	SD	R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU	R1,R1,#8	2	3		4	Execute 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

Thread level parallelism

- ILP is used in straight line code or loops
- Cache miss (off-chip cache and main memory) is unlikely to be hidden using ILP.
- Thread level parallelism is used instead.
- Thread: process with own instructions and data
  - thread may be a process part of a parallel program of multiple processes, or it may be an independent program
  - Each thread has all the state (instructions, data, PC, register state, and so on) necessary to allow it to execute



From: Tullsen, Eggers, and Levy,

"Simultaneous Multithreading: Maximizing On-chip Parallelism, ISCA 1995.

Thread Level parallelism

- Multithreading: multiple threads to share the functional units of 1 processor via overlapping
  - processor must duplicate independent state of each thread e.g., a separate copy of register file, a separate PC, and for running independent programs, a separate page table
  - memory shared through the virtual memory mechanisms, which already support multiple processes
  - HW for fast thread switch; much faster than full process switch ≈ 100s to 1000s of clocks
- When to switch?
  - Alternate instruction per thread (fine grain)
  - When a thread is stalled, perhaps for a cache miss, another thread can be executed (coarse grain)

Fine-Grained Multithreading

- Switches between threads on each instruction, causing the execution of multiples threads to be interleaved
- Usually done in a round-robin fashion, skipping any stalled threads
- CPU must be able to switch threads every clock
- Advantage is it can hide both short and long stalls, since instructions from other threads executed when one thread stalls
- Disadvantage is it slows down execution of individual threads, since a thread ready to execute without stalls will be delayed by instructions from other threads
- Used on Sun's T1

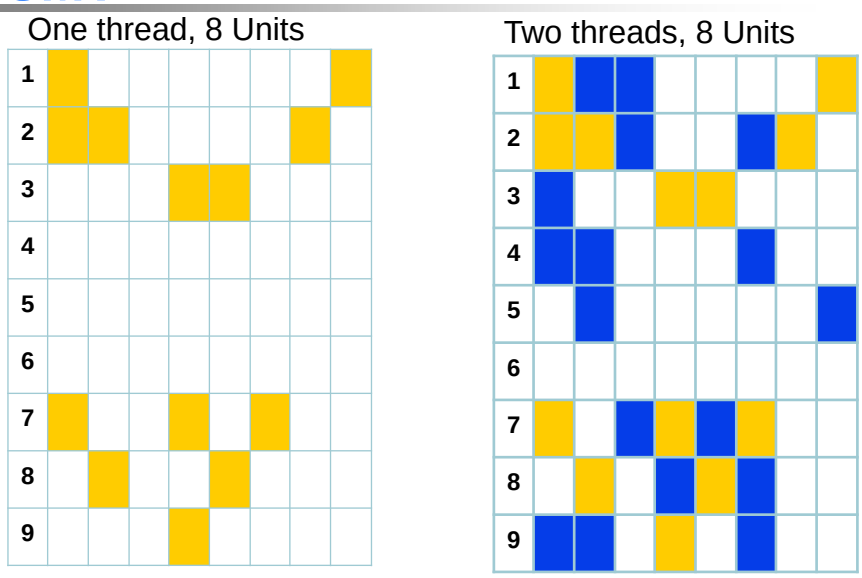
Coarse-Grained Multithreading

- Switches threads only on costly stalls, such as L2 cache misses
- Advantages
  - Need to have very fast thread-switching
  - Doesn't slow down thread, since instructions from other threads issued only when the thread encounters a costly stall
- Disadvantage is hard to overcome throughput losses from shorter stalls, due to pipeline start-up costs
  - Since CPU issues instructions from 1 thread, when a stall occurs, the pipeline must be emptied or frozen
  - New thread must fill pipeline before instructions can complete
- Because of this start-up overhead, coarse-grained multithreading is better for reducing penalty of high cost stalls, where pipeline refill << stall time

Simultanuous Multithreading SMT

- Fine-grained multithreading implemented on top of multiple-issued dynamically scheduled processor.
- Multiple instructions from different threads.

SMT



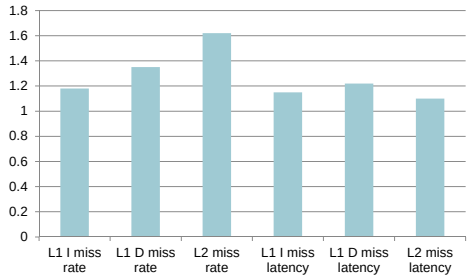
Multithreading

Sun T1

- Focused on TLP rather than ILP
- Fine-grained multithreading
- 8 cores, 4 threads per core, one shared FP unit.
- 6-stage pipeline (similar to MIPS with one stage for thread switching)
- L1 caches: 16KB I, 8KB D, 64-byte block size (misses to L2 23 cycles with no contention)
- L2 caches: 4 separate L2 caches each 750KB. Misses to main memory 110 cycles assuming no contention

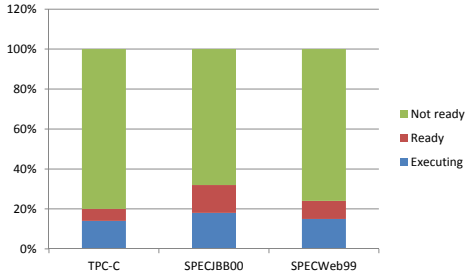
Sun T1

- Relative change in the miss rate and latency when executing one thread per core vs 4 threads per core (TPC-C)



Sun T1

- Breakdown of the status on an average thread. Ready means the thread is ready, but another one is chosen – The core stalls only if all the 4 threads are not ready



Sun t1

- Breakdown of the causes for a thread being not ready