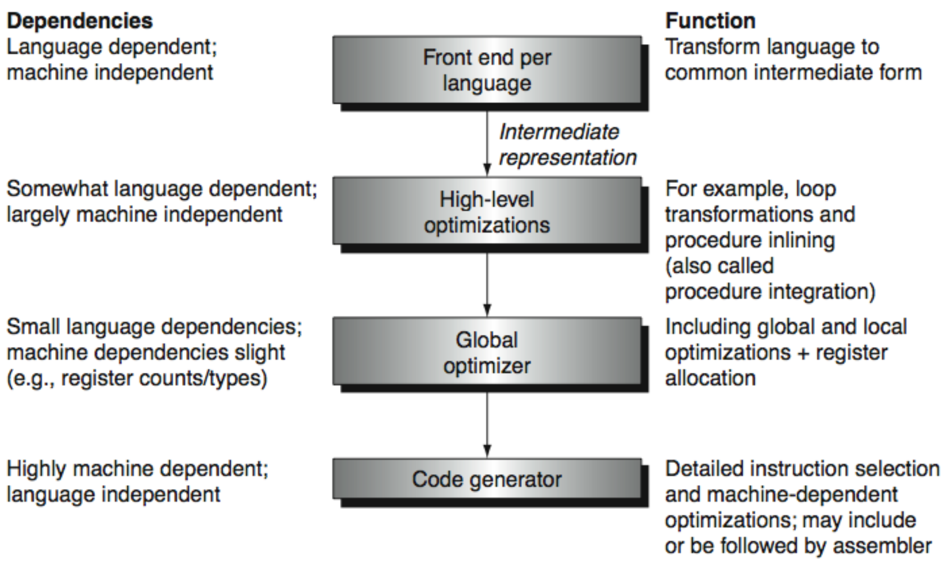
Chapter 6: Advanced Pipelining

(class text - Fig 2.31, pg. 116)

Recall:



Compiler Scheduling for Instruction Parallelism

• to improve performance further, we need to extract more "instruction level parallelism"; Recall

pipeline CPI = Ideal CPI (1) + structural stalls* + data hazard stalls* + control stalls*

(* = average stalls per instruction)

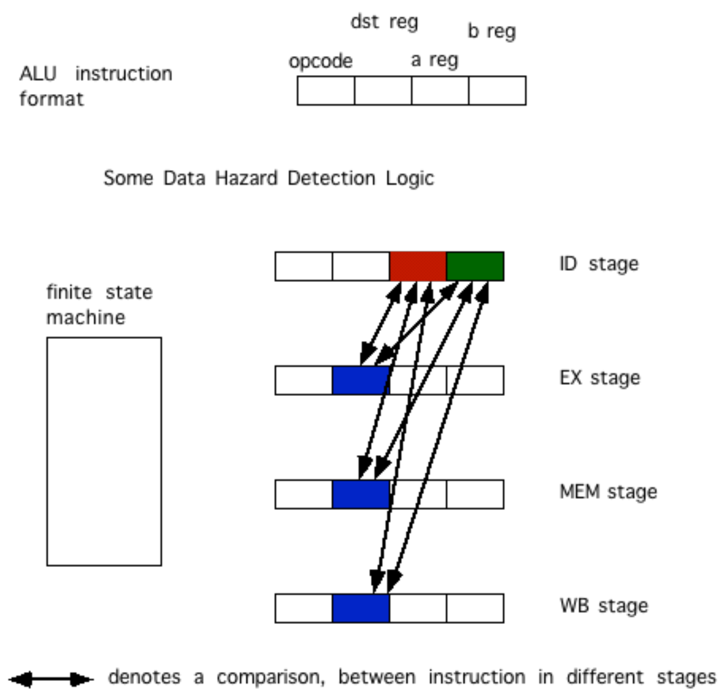
• for now, we assume the following latencies in the TABLE below (pg 304, ref text); (these change according to the design of each pipelined machine)

inst. producing result	inst. consuming result	latency in cc
FP ALU op	another FP ALU op	3 cc
Load double	FP ALU op	1 cc
FP ALU op	store double / FP	2 cc
Load double	store double	0 cc

• in this context, **latency = # of <u>extra cc</u> after the producing instruction enters its EX stage, before the result can be used,** which is equal to the # of cc needed to avoid data hazard stalls between the producing and consuming instructions

• Note: this lecture material is not covered in the class textbook, so take notes

Comparison Hardware to Detect Data hazards in ID Stage



Technique # 1: Loop Unrolling

- consider a loop to add a scalar "s" to a vector:

for (i=1000; i > 0; i=i-1)
x[i] = x[i] + s;

- before "unrolling" the loop, the unscheduled code looks like this:

```
Loop:   LD      F0, 0(R1)      ; load element x[i] into F0
        ADDD    F4, F0, F2     ; add scalar in F2 to F0, store in F4
        SD      0(R1), F4      ; store result back to memory
        SUBI    R1, R1, #8     ; dec. pointer by 8 bytes = size of doubleFP
        BNEZ    R1, Loop
```

Regular Loop, Unscheduled

(ref text - pp. 305)

- the "unscheduled" code executes like this:

Loop:	LD	**F0**, 0(R1)	1 cc	
	stall		2 cc	
	ADDD	**F4**, **F0**, F2	3 cc	; 1 cc stall from table 1
	stall		4 cc	
	stall		5 cc	
	SD	0(R1), **F4**	6 cc	; 2 cc stall from table 1
	SUBI	**R1**, R1, #8	7 cc	
	stall		8 cc	; the earliest forward still creates a stall
	BNEZ	**R1**, Loop	9 cc	
	no-op		10 cc	; branch delay slot unused

- 10 cc per vector element. At a 1 GHz clock, performance = 100 MFLOP / sec
- 1 MFlop = 1 million floating point operations per second
- MFLOP rating = (1 GHz)*(1 FLOP/10cc) = 100 MFLOP/sec

Timing Diagram - Regular Loop - Branch Hazard

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
LD F0,0(R1)	F	D	E	M	WB									
ADDD F4,F0,F2		F	**D**	s	**A1**	A2	A3	**A4**	M	**WB**				
SD 0(R1),F4			F	s	D	**EX**	s	s	**M**	**WB**				
SUBI R1,R1,#8				s	F	D	s	s	EX	M	WB			
BNEZ R1,Loop						F	s	s	D	s-D	EX	M	WB	
No-Op							s	s	F	s	D	EX	M	WB

- The events in cc 9 are interesting:
- The BNEZ instruction moves into the ID unit, but it cant make a decision since R1 is being computed in the EX stage
- we have 2 options:
- => (1) stall the BNEZ in the ID stage, add forwarding hardware from the EX stage back into the ID stage, and resolve the BNEZ instruction in next cc (which introduces a stall)
- (2) don't stall the BNEZ in the ID stage, add forwarding hardware from the EX stage back into the ID stage, and resolve the BNEZ instruction in the same cc (which stretches the clock)
- lets choose the 1st option, to avoid stretching the clock
- here, we choose to allow 2 instructions into the MEM stage, and WB stage, per cc

Timing Diagram - Regular Loop - ME,WB Hazard

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	
LD F0,0(R1)	F	D	E	M	WB										
ADDD F4,F0,F2		F	D	s	A1	A2	A3	A4	M	WB					
SD 0(R1),F4			F	s	D	EX	s	s	s	M	WB				
SUBI R1,R1,#8				s	F	D	s	s	s	EX	M	WB			
BNEZ R1,Loop						F	s	s	s	D/s	D	EX	M	WB	
No-Op									s	s	F	D	EX	M	WB

- Here we assume the MEM and WB units accept only 1 instruction per cc
- The events in cc 9 are interesting:
- in this example, to avoid adding hardware, we chose to keep a structural hazard: we only let one instruction into the MEM stage, or the WB stage, per clock cycle
- however, now we have to add forwarding hardware from the MEM stage, back intro the MEM stage: see clock cycle 9-10: the ADDD instruction is in the MEM stage, and it needs to supply F4 to the SD, which will enter the MEM stage in clock cycle 10
- given that we need to add this forwarding hardware in this example, it may be better to allow 2 instructions into the MEM unit and WB unit per cc

Regular Loop, Compiler Scheduled

(ref text - pp. 306)

Loop:	LD	**F0**, 0(R1)	1 cc	
	SUBI	R1, R1, #8	2 cc	
	ADDD	**F4**, **F0**,F2	3 cc	
	stall		4 cc	; this stall caused by SD
	BNEZ	R1, Loop	5 cc	
	SD	**F4, 8(R1)**	6 cc	; move SD into branch delay slot

- here is a version of the same loop, after ‘**compiler scheduling**’
- by moving SD into the branch delay slot, 2 stalls are eliminated
- SUBI moved earlier, and memory address (relative to R1) adjusted by compiler
- performance = 1 FLOP / 6 cc
- at 1 GHz clock, performance =(1 GHz)*(1 FLOP/6 cc) = 167 MFLOPS / sec
- a 67 % improvement over the unscheduled code (100 MFLOP performance)
- to expose even more parallelism, compiler can “unroll” loop to expose 4 iterations at once

Loop Unrolled 4 Times, Unscheduled (ref text - pp. 307)

```
Loop:            LD     F0, 0(R1)       ; 1st element into F0
        1 stall  ADDD   F4, F0, F2
        2 stalls SD     0(R1), F4
                 LD     F6,-8(R1)       ; 2nd element into F6
        1 stall  ADDD   F8, F6, F2
        2 stalls SD     -8(R1), F8
                 LD     F10,-16(R1)     ; 3rd element into F10
                 ADDD   F12, F10, F2
                 SD     -16(R1), F12
                 LD     F14,-24(R1)     ; 4th element into F14
        1 stall  ADDD   F16, F14, F2
        2 stall  SD     -24(R1), F16
                 SUBI   R1, R1, #32     ; dec pointer by 4 double FPs = 32 bytes
        1 stall  BNEZ   R1, Loop
                 branch delay slot
```

- **without scheduling**, many stalls : (3 inst + 3 stalls per element) * 4 elements + 2 loop control + 1 branch delay slot + 1 stall = 28 cc per pass; performance = (1 GHz clock)*(4 FLOPS/28 cc) = 143 MFLOPS /sec (Effective rate is now slower than scheduled single loop iteration on last slide = 167 MFLOP/sec)

Loop Unrolled 4 Times, Scheduled (ref text - pp. 308)

```
Loop:   LD     F0, 0(R1)       ; 1st element into F0
        LD     F6,-8(R1)       ; 2nd element into F6
        LD     F10,-16(R1)     ; 3rd element into F10
        LD     F14,-24(R1)     ; 4th element into F14
        ADDD   F4, F0, F2
        ADDD   F8, F6, F2
        ADDD   F12, F10, F2
        ADDD   F16, F14, F2
        SD     0(R1), F4
        SD     -8(R1), F8
        SUBI   R1, R1, #32     ; dec pointer by 4 double FP #s
        SD     +16(R1), F12
        BNEZ   R1, Loop
        SD     +8(R1), F16
```

- with **scheduling**, (4 FLOP/14 cc); effective rate of (1 FLOP/3.5 cc)
- at 1 GHz clock, performance = (1 GHz)*(1 FLOP/3.5 cc) = 286 MFLOP / sec
- much faster than all prior methods !

Loop Unrolled 4 Times, Scheduled

```
Loop:   LD      F0, 0(R1)       ; 1st element into F0
        LD      F1,-8(R1)       ; 2nd element into F1
        LD      F2,-16(R1)      ; 3rd element into F2
        LD      F3,-24(R1)      ; 4th element into F3
        ADDD    F4, F0, F31     ; scalar now in F31
        ADDD    F5, F1, F31
        ADDD    F6, F2, F31
        ADDD    F7, F3, F31
        SUBI    R1, R1, #32     ; dec pointer by 4 double FP #s (32 bytes)
        SD      +32(R1), F4     ; adjust effective address by +32
        SD      +24(R1), F5
        SD      +16(R1), F6
        BNEZ    R1, Loop
        SD      +8(R1), F16
```

• here is another way of writing the scheduled code. We used registers in a linear order (F0, F1, f2 etc, and we put the scalar constant in F31)

Hardware Cost of Previous Slide

Instructions		Clock Cycle																									Comment
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	
foo: L.D	F0, 0(R1)	F	D	E	M	W																					
L.D	F1, -8(R1)		F	D	E	M	W																				
L.D	F2, -16(R1)			F	D	E	M	W																			
L.D	F3, -24(R1)				F	D	E	M	W																		
ADD.D	F4, F0, F31					F	D	A1	A2	A3	A4	Mi	W														
ADD.D	F5, F1, F31						F	D	A1	A2	A3	A4	Mi	W													
ADD.D	F6, F2, F32							F	D	A1	A2	A3	A4	Mi	W												
ADD.D	F7, F3, F31								F	D	A1	A2	A3	A4	Mi	W											
SUBI	R1,R1,#32									F	D	E	M	W													2 inst in M stage
S.D	+32(R1), F4										F	D	E	M	Wi												2 inst in M, W stages
S.D	+24(R1), F5											F	D	E	M	Wi											2 inst in M, W stages
S.D	+16(R1), F6												F	D	E	M	Wi										2 inst in W stage
BNEZ	**R1, foo**													F	D	E	M	W									
S.D	+8(R1), F7														F	D	E	M	Wi								

14 cc per loop iteration

(1) To achieve this performance, we must allow up to 2 instructions to be in M stage and W stage in 1 cc
(2) The ADD.D instructions pass through the MEM stage by assumption, but they do not access the Data-Memory-cache
Therefore, lets write 'Mi' in the space-time diagram, to denote 'M idle', ie these instructions don't access the data cache
(3) The SD instructions pass through the Wrie-back stage, but they do not write back to any register in the ID stage.
Therefore, lets write 'Wi' in the space-time diagram, to denote 'W idle', ie these instructions do not perform a write-back

• to achieve the performance of 14 cc, observe that the M and W stages must be '**widened**' to allow 2 instructions to enter and exit per cc. We must add extra hardware to allow this.

• Also observe that in the MEM stage, only 1 instruction ever Reads or Writes to Memory per cc, which is good (the data-cache does not have to be widened).

Summary, Loop Unrolling and Scheduling

- regular loop, unscheduled and with all stalls : **10 cc per FLOP, 100 MFLOP/s**
- regular loop, compiler scheduled to reduce stalls : **6 cc per FLOP, 167 MFLOP/s**
- unrolled loop, unscheduled and with all stalls : **6.8 cc per FLOP, 143 MFLOP/s**
- unrolled loop, compiler scheduled to reduce stalls : **3.5 cc per FLOP, 286 MFLOP/s**

- **Observations**: Unrolling to loop to expose more parallelism can improve performance, when the compiler schedules the code to reduce / eliminate stalls
- the key to improving performance is doing multiple loads early, so that FP ops can proceed without stalls once data is loaded
- to avoid **data hazards**, the compiler must use many FP registers; basically each unrolled loop iteration uses 2 new FP registers to store its results without data hazards (if we used the same FP registers as other loop iterations, we would have many data hazards and stalls)
- one limitation to loop unrolling is the number of FP registers required
- **Q: if we had an infinite # of FP registers, what is the best performance of the unrolled loop ? (Answer: 3 instructions per vector element).**

Notes

Static Multiple-Issue Pipelines

(class text -section 6.9, pg 433)

- In a "**static multiple issue**" pipeline, n instructions **issue together** in each clock cycle
- in an ideal n-issue machine, effective CPI = 1/n < 1 !
- all instructions that issue together are arranged in "**Issue-Slots**" and together form an "**Issue-Packet**" (pg 436) (like race horses which are given starting slots)
- usually, there is a restriction on which types of instructions can fit into an Issue-Packet
- If an Issue-Slot cannot be filled with a useful instruction due to hazards, a **No-Op** is inserted into the slot
- This architecture is called by several names, "Static Multiple Issue", or "SuperScalar", or "Very Long Instruction Word" (VLIW), or "Explicitly Parallel Instruction Computer" (EPIC - Intel's name)
- the first implementations of the **Intel IA-64** architecture, the Itanium-1 and Itanium-2, are static multiple issue machines, called 'EPIC' by Intel

Some Existing Multiple-Issue Machines

(ref text - fig. 3.23, pg 216)

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	dynamic	hardware	static	in-order execution	Sun UltraSPARC II/III
Superscalar (dynamic)	dynamic	hardware	dynamic	some out-of-order execution	IBM Power2
Superscalar (speculative)	dynamic	hardware	dynamic with speculation	out-of-order execution with speculation	Pentium III/4, MIPS R10K, Alpha 21264, HP PA 8500, IBM RS64III
VLIW/LIW	static	software	static	no hazards between issue packets	Trimedia, i860
EPIC	mostly static	mostly software	mostly static	explicit dependences marked by compiler	Itanium

Figure 3.23 The five primary approaches in use for multiple-issue processors and the primary characteristics that distinguish them. This chapter has focused on the hardware-intensive techniques, which are all some form of superscalar. The next chapter focuses on compiler-based approaches, which are either VLIW or EPIC. Figure 3.61, near the end of this chapter, provides more details on a variety of recent superscalar processors.

(We'll study **dynamic** multiple-issue machines in a few lectures. They are quite different from static multiple-issue machines.)

A Dual-Issue MIPS Processor

(class text -section 6.9, pg 433)

• consider a simple "static multiple issue" pipeline: 2 instructions issued per cc

• **RESTRICTION**: • 1 slot reserved for ALU or BRANCH instructions,
• 1 slot reserved for LOAD or STORE instructions

• these restrictions simplifies ID stage, since they eliminate the hazards that might appear if multiple instructions of any type where allowed to issue together

• also, not too much more hardware is required, since we have an integer ALU pipeline stage which can be now be kept busy **in parallel** with the MEM pipeline stage anyway

• Recall our **Basic Loop** to add scalar in F31 to a vector in memory (see pg 438 class text)

```
Loop:   LD      F0, 0(R1)       ; fetch array element
        ADDD    F4, F0, F31     ; add FP scalar
        SD      0(R1), F4       ; store array element
        SUBI    R1, R1, #8
        BNEZ    R1, Loop
```

• this code sequence benefits slightly with dual issue, and with loop unrolling it will benefit alot

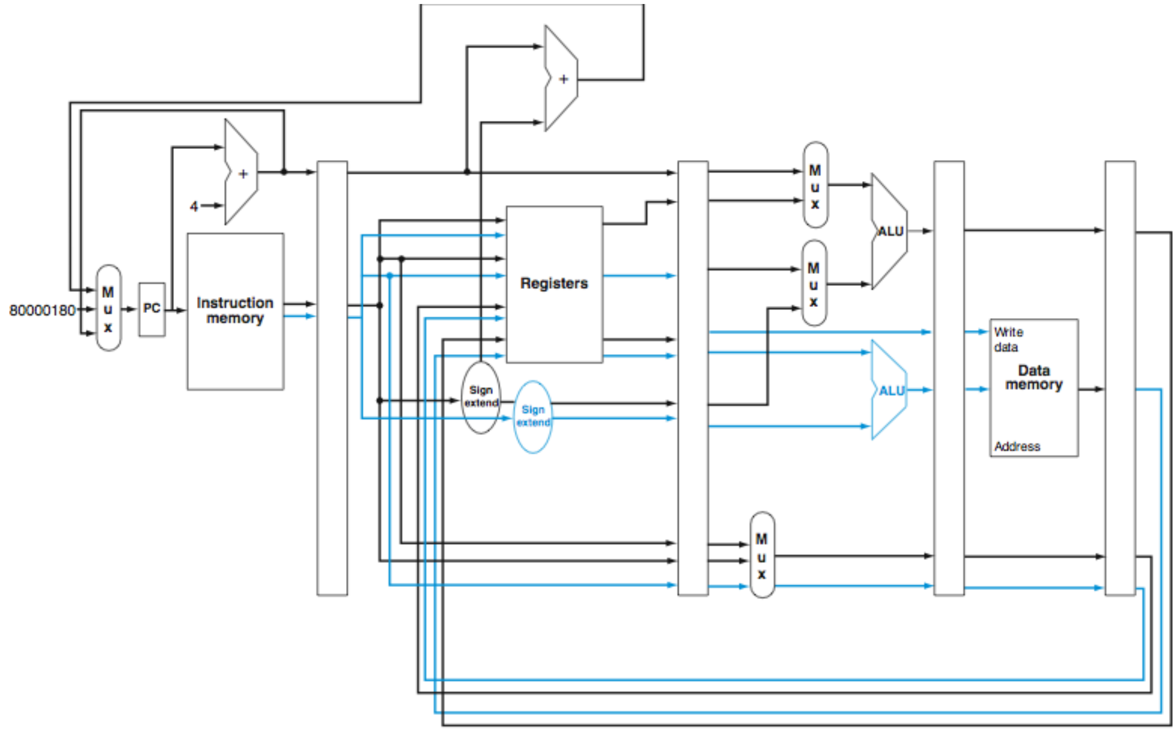
Space-Time Diagram (class text - fig. 6.44, pp. 433)

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

FIGURE 6.44 Static two-issue pipeline in operation. The ALU and data transfer instructions are issued at the same time. Here we have assumed the same five-stage structure as used for the single-issue pipeline. Although this is not strictly necessary, it does have some advantages. In particular, keeping the register writes at the end of the pipeline simplifies the handling of exceptions and the maintenance of a precise exception model, which become more difficult in multiple-issue processors.

• dual-issue significantly increases rate of LOAD/STORE issues, but requires separate caches for instructions (for the IF stage) and data (for the MEM stage)

2-Issue MIPS Datapath (class text - fig. 6.45, pp. 437)



No Unrolling, Just Scheduled on Dual-Issue Machine

- In this example, assume: (1) a 4-stage FP-ADD pipeline, (2) we can let 2 instructions into the MEM unit per clock cycle, provided only 1 instruction accesses MEM, (3) we can let 2 instructions into the WB unit per clock cycle
- **EXAMPLE #1: No Unrolling**

LD/SD slot	Others slot	Comment
LD F0,0(R1)	no-op	
no-op	SUB1 R1, R1, #8	
no-op	ADDD **F4**,F0,F31	
no-op	BNEZ R1,loop	
stall		ID unit inserts 1 stall
SD **F4**,+8(R1)	no-op	data hazard on F4

No Unrolling, Just Scheduled on Dual-Issue Machine

	1	2	3	4	5	6	7	8	9	10	11	12	13	14		
LD F0,0(R1)	F	D	E	M	WB											
NO-OP	-	-	-	-	-											
NO-OP		-	-	-	-	-										
SUBI R1,R1,#8		F	D	E	M	WB										
NO-OP			-	-	-	-	-									
ADDD F4,F0,F31			F	D	**A1**	**A2**	**A3**	**A4**	M	WB						
NO-OP				-	-	-	-	-								
BNEZ R1,Loop				F	D	E	M	WB								
SD F4,0(R1)					F	D	E	**s**	M	WB						
NO-OP					-	-	-	**s**	-	-						
LD F0,0(R1)						F	D	**s**	E	M	WB					
NO-OP						-	-	**s**	-	-	-					
NO-OP							-	**s**	-	-	-	-				
SUBI R1,R1,#8							F	**s**	D	E	M	WB				
NO-OP									-	-	-	-	-			
ADDD F4,F0,F31									F	D	**A1**	**A2**	**A3**	**A4**	M	WB
NO-OP										-	-	-	-	-		
BNEZ R1,Loop										F	D	E	M	WB		
SD F4,0(R1)											F	D	E	**s**	M	WB
NO-OP											-	-	-	**s**	-	-

• here is the timing diagram: Note that we must add extra hardware so that No-Op instructions do not enter the MEM or WB stages, otherwise there would be 3 instructions in the MEM unit in clock cycle 10

Unrolled 4 times and Scheduled on Dual-Issue Machine

(Example completed in class)

Example 1: dual issue, unrolled 4 times & scheduled

Restriction	LD or SD slot	Others slot (ALU,FP,BRA)	comment / hazard
loop	LD F0,0(R1)	no-op	
	LD F1,-8(R1)	SUBI R1,R1,#32	
	LD F2,+16(R1)	ADDD F4,F0,F31	
	LD F3,+8(R1)	ADDD F5,F1,F31	
	no-op	ADDD F6,F2,F31	
	no-op	ADDD F7,F3,F31	
	SD F0,+32(R1)	no-op	
	SD F1,+24(R1)	no-op	
	SD F2,+16(R1)	BNEZ R1,loop	
	SD F3,+8(R1)	no-op	branch delay slot

- in 10 cc, there are 20 slots and 6 no-ops: the slot-usage efficiency is 14/20 = 70%, and there are no stalls: 4 FLOPs take 10 cc, for an effective MFLOP rate of (1 GHz) *(4 Flops/10cc) = 400 MFLOP/sec.
- Dual-Issue is about 50 % faster than best scheduled single issue (286 MFLOP/sec)

Unrolled 8 times and Scheduled on Dual-Issue Machine

(Example completed in class)

Example 1: dual issue, unrolled 8 times & scheduled					
Restriction	LD or SD slot		ALU or Branch slot		comment/hazard
loop	LD	F0,0(R1)	no-op		
	LD	F1,-8(R1)	SUBI	R1,R1,#64	
	LD	F2,+48(R1)	ADDD	F8,F0,F31	adjust address offset
	LD	F3,+40(R1)	ADDD	F9,F1,F31	
	LD	F4,+32(R1)	ADDD	F10,F2,F31	
	LD	F5,+24(R1)	ADDD	F11,F3,F31	
	LD	F6,+16(R1)	ADDD	F12,F4,F31	
	LD	F7,+8(R1)	ADDD	F13,F5,F31	
	SD	F8,+64(R1)	ADDD	F14,F6,F31	
	SD	F9,+56(R1)	ADDD	F15,F7,F31	
	SD	F10,+48(R1)	no-op		
	SD	F11,+40(R1)	no-op		
	SD	F12,+32(R1)	no-op		
	SD	F13,+24(R1)	no-op		
	SD	F14,+16(R1)	BNEZ	R1,loop	
	SD	F15,+8(R1)	no-op		fill branch delay slot

- effective MFLOP rate of (1 GHz)*(8 Flops/16cc) = 500 MFLOP/sec.
- Dual-Issue is about 180 % faster than best scheduled single issue (286 MFLOP/sec)

Notes

Static Multiple-Issue - Observations

- recall in 5-stage MIPS pipeline, LOAD had a latency of 1 cc, ie next inst. cannot use result of LOAD without stalling for 1 cc
- in a dual-issue MIPS, up to 3 successive instructions cannot use result of LOAD without stalling - this limits parallelism
- in the dual-issue MIPS, 1cc branch delay slot == 2 instruction slots
- in superscalar machines, load stalls and branch delay slots represent larger penalties
- a dual-issue CPU will only issue 2 instructions if they fit within the Issue Slots, otherwise it will issue them sequentially (by inserting NO-OPs into slots)
- superscalar processors are “backwards” compatible, and can execute sequential and unscheduled code too, but with lower performance
- Apple computer relied upon backwards compatibility when it changed its processor platform, from the Motorola 68,000 family of CISC machines to the the PowerPC superscalar RISC family; most of the original software for the PowerPC was the serial 68,000 style of code which the multiple-issue PowerPC had to execute
- Apple established that such a transition could be accomplished without a major setback in the business operation, which perhaps motivated Intel and HP to embark on their joint project to develop a new processor architecture for their products (**Itanium-EPIC**)

A Dual- Issue MIPs with - Integer and FP Parallelism

(ref text)

- consider another **slightly different** dual-issue pipeline with 2 instructions issued per cc:
- **RESTRICTION:**
- 1st Issue-Slot is for LD, SD, and all Integer instructions including Branches
- 2nd Issue-Slot is for all FP instructions only
- This restriction doesn’t appear to change the performance, from the examples I have looked at.
- In some textbooks such as the class reference textbook, you may see this type of dual-issue pipeline described
- The Itanium processor, which we will talk about next, lets us change the restrictions on the Issue-Slots.

Dynamically Scheduled Multiple-Issue Machines

• In a **Statically Scheduled machine**, the compiler schedules all instructions to avoid hazards: the ID unit simply checks to see if instructions can issue together without hazards, otherwise the ID unit inserts stalls until the hazards clear

• A future section (Dynamic Scheduling in chapter 3/4) will deal with **Dynamically Scheduled machines**, where **hardware-based** techniques are used to detect hazards, to re-schedule the instructions 'on-the-fly' and issue them in the right order, and improve performance

• Dynamic scheduling is used in the Pentium III and 4, the Athlon, the MIPS R10000, the SUN UltraSPARC III; the PowerPC 603, PowerPC G3, PowerPC G4, and the Alpha 21264

• In contrast, **static multiple-issue** with compiler-based scheduling is used in the Intel IA-64 **Itanium** architectures

• In 2007, the dual-core and quad-core Intel laptop processors use the Pentium 5 family of dynamically scheduled processors. The Itanium is primarily used in servers.

The Itanium Processor – Historic Issues

• The RISC revolution in the 1990s caught many companies off-guard; The company 'Digital Equipment Corporation' (DEC), which manufactured the refrigerator size DEC-Vax computer and which was probably the biggest computer company at the time, was eliminated, wiping out billions of dollars of market capitalization (see next slide)

• To avoid the same fate, Intel and HP embarked on a 'secret' project to develop a 64-bit *super-chip for the future*, to replace the older 32-bit Pentium-family of processors

• The code-named the new chip the 'Itanium' and it was released in 2001

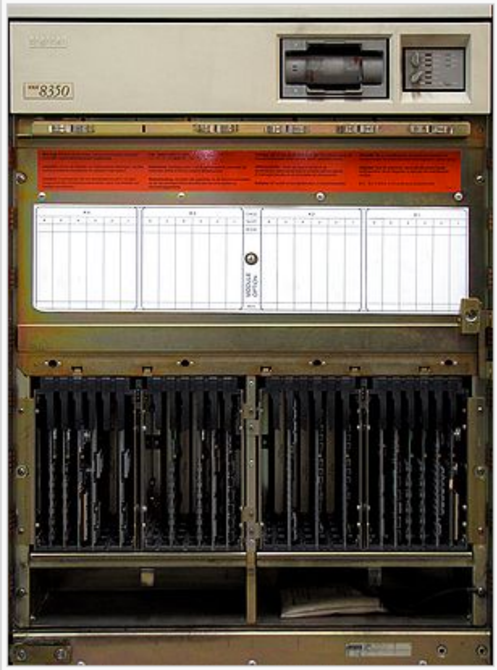
• Unfortunately, the Itanium software was incompatible with the existing library of 30 years of Pentium code

• The competitor company AMD then shipped a 64-bit version of the Pentium-family, called the Opteron.

• The computing industry largely moved to the Opteron processor, which could run Pentium-style code and kept alive the existing library of 30 years of Pentium code, thereby finishing-off the Itanium vision painted by Intel and HP

• The Itanium is still used in a niche market of proprietary high-end servers manufactured by HP and a few other Itanium supporters, but it failed to live up to the promise presented by Intel and HP

Photo – DEC Vax Machine – Historic Issues



VAX 8350 front view with cover removed.

The Itanium Processor

• the Intel IA-64 architecture is a RISC-style **static multiple-issue machine**:

• instructions grouped into "**Bundles**", each 128 bits wide with 3 instructions, and all 3 instructions in a bundle must issue together or stall together

• a 5 bit vector for each instruction identifies which of 5 different execution units are required (INT-ALU, NON-INT-ALU, MEM UNIT, FP UNIT, BRANCH UNIT)

• **PREDICATION**: The IA-64 includes "Predication", a technique to eliminate many branches and branch hazards by making the execution of an instruction depend upon the contents on a "Predicate Register"

• the code sequence "if (p) (statement 1) else (statement 2)" normally requires 2 branches, one after (p) and one after (statement 1)

• with Predication, it is replaced by

(p) statement 1

(~p) statement 2

• each line is executed if the predicate at the beginning is true, otherwise it becomes a **No-Op**

• the original branch code sequence is replaced by 2 lines of code, with no branches

• nearly every instruction can be predicated, and there are many predication registers to store predicates

Itanium Processor - Registers

(ref text - pg. 351)

- IA-64 Itanium-1 adds many more registers:
- there are 128 64-bit INT registers (actually 65 bits wide)
- there are 128 82-bit FP registers (2 bits more than the IEEE standard 80 bits)
- 8 64-bit BRANCH registers, to store branch target addresses for indirect branches

- the 128 INT registers are allocated to support parameter passing for procedure calls: the lower 32 registers are always visible in any procedure and are accessed as R0 .. R31
- the remaining 96 registers can be partitioned into "frames" , where a procedure call can be allocated a frame of registers to store parameters - avoids the need to pass parameters through a stack
- parameters can be passed to and from the procedure call using the registers in its frame
- each procedure has a register called the "current frame pointer", to point to where its frame starts
- the machine is 3 instructions (slots) wide, the 3 instructions are called a 'BUNDLE', and all 3 instructions in a BUNDLE must issue together or stall together

Itanium-1 & Itanium-2 Processors

(class text - Fig. 6.48. pg 442)

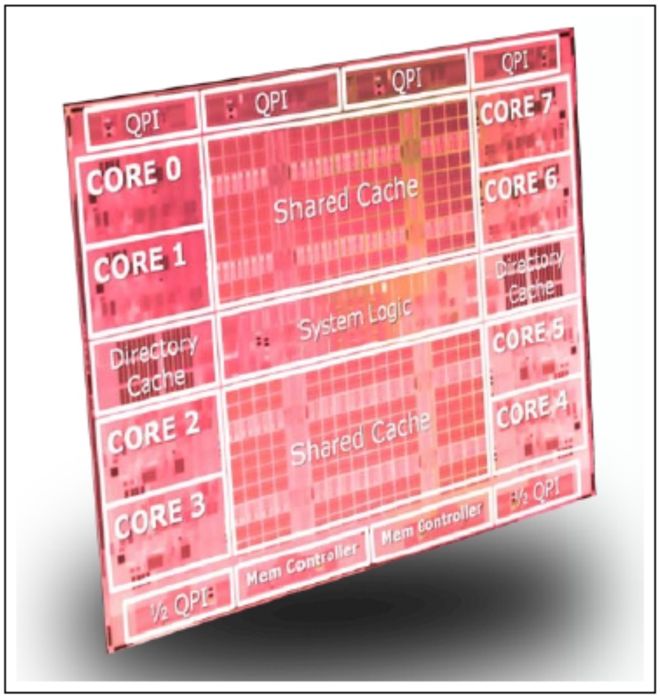
Processor	Maximum instr. issues / clock	Functional units	Maximum ops. per clock	Max. clock rate	Transistors (millions)	Power (watts)	SPEC int2000	SPEC fp2000
Itanium	6	4 integer/media 2 memory 3 branch 2 FP	9	0.8 GHz	25	130	379	701
Itanium 2	6	6 integer/media 4 memory 3 branch 2 FP	11	1.5 Ghz	221	130	810	1427

FIGURE 6.48 A summary of the characteristics of the Itanium and Itanium 2, Intel's first two implementations of the IA-64 architecture. In addition to higher clock rates and more functional units, the Itanium 2 includes an on-chip level 3 cache, versus an off-chip level 3 cache in the Itanium.

- According to class text, the latest Itaniums can issue 2 Bundles (6 instructions) per cc
- The 2005 Itanium-2 uses **221 Million transistors**, and dissipates **130 Watts**
- take note: It has 6 INT ALU units, 2 pipelined FP units, 4 MEM units, to help avoid structural hazards, so that we can issue bundles without waiting

Itanium – Poulson Die (2010-2011)

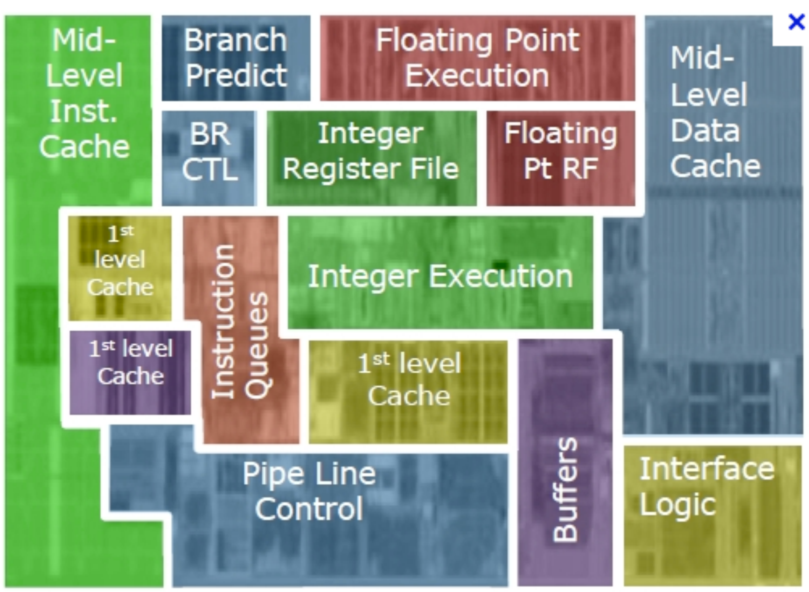
(www.theregister.co.uk/2011/02/20/intel_poulson_itanium_isscc)



Intel's future "Poulson" Itanium server processor

Itanium - Poulson Core (2010-2011)

(www.theregister.co.uk/2011/02/20/intel_poulson_itanium_isscc)



Itanium - Poulson Summary (2010-2011)

(www.theregister.co.uk/2011/02/20/intel_poulson_itanium_isscc)

The Poulson chip is the ninth in the Itanium family, which is more than a decade old and which was intended to replace x86 processors in the glorious future painted by Intel, HP, and the other Itanium partners back in the mid-to-late 1990s. That didn't happen, obviously, and everyone else that had adopted Itanium beside HP has backed Intel's high-end Xeon 7500s for everything but the proprietary platforms they don't want to move to another chip architecture. Again.

The Poulson chip has eight cores, two directory caches, five QuickPath Interconnect (QPI) links, two memory controllers, two shared L2 caches, and a bunch of system logic all on the same piece of silicon. It weighs in at 3.1 billion transistors, and is 588 square millimeters in size. The current Tukwila Itanium chip, by comparison, has four cores, a total of 2 billion transistors, and is 700 square millimeters in area. The double shrink from 65 to 32 nanometers allows for a lot more stuff to be crammed onto the chip, and also a reduction of the size of the chip by about 20 per cent and a slight reduction in the thermal design point, which drops from 185 watts with top-end Tukwila parts to 170 watts with the fastest Poulson parts.

Itanium Compiler Scheduling (ref text - ch. 4.7, pg. 351)

- the compiler can create the code using 2 strategies:
- (#1) **minimize the number of bundles**; pack 3 instructions per '**bundle**' whenever possible, knowing that many bundles will stall due to data hazards or structural hazards
- (#2) **minimize the number of stalls**; The compiler inserts **NO-OPs** into slots when needed, and arranges the bundles to execute without stalling if possible (compiler-based scheduling)
- Note that the Itanium is a **multiple-issue linear pipeline machine**; the ID (issue) stage controls access to all the Execution units, and stalls all issues if necessary
- Lets consider compiler scheduling using the 2 methods. The loop **x[I] = x[I] + s** is unrolled 7 times, and scheduled (in the class textbook): here are the reported results:
- (a) strategy #1 - minimize the number of instruction bundles
- (b) strategy #2 - minimize the number of stalls (blank slots represent **NOOPs**).
- Strategy #1 yields 9 bundles, and executes in 21 clock cycles for 7 iterations of the loop.
- Strategy #2 yields 11 bundles and executes in 12 clock cycles for 7 iterations of the loop.

- lets repeat these examples in our class notes, but we'll only unroll the loop 4 times (next slides)

▪ In general, using strategy #2 (maximize code density), we really should maximize code density as a first criterion, and and then maximize performance (minimize stalls) when possible.

▪ In general, using strategy #1 (maximize code speed), we really should maximize code speed as a first criterion, and and then maximize code density when possible.

Itanium Bundle Restrictions

- the 2 Itanium coding examples on the next slide are from the reference textbook (pg 355), and there is a limited explanation on them
- The Itanium places restrictions on which types of instructions can go in each slot. Some of the instruction types are: M=Memory, I=Integer ALU, FP = Floating Point, B = branch.
- The Itanium allows 29 different combinations of instruction types to go into a bundle, and these **restrictions** are called 'templates' by Intel
- For example, in the template column, " M M I" denotes (2 memory instructions, one Integer instruction)
- Common Itanium templates are: "MMI", "MMF", "MII", "MFI", "MIB" ,'BBB",
- " MMB", "MFB"
- Interestingly, the above templates **do not** allow 2 FP operations per bundle (which would limit FP performance)
- Equally interestingly, the above templates **do allow** 3 branch instructions to be placed in one bundle. It would be difficult to decide which branch takes precedence if all three branches were taken simultaneously

Example - Itanium Scheduling (pg 356, ref text)

• Lets work out an example in class

• **TYPICAL ASSUMPTIONS:**

• (1) there are 2 MEM units, each 2 stage pipeline

• (2) 2 pipelined FP ADD units, each 3 stage pipeline

• (3) 1 pipelined FP MULT unit, 6 stage pipeline

• (4) **any 3** non-branch instructions can be placed in one bundle if there is enough hardware (They all issue together or they all stall together)

• (5) **at most one branch** can be placed in a bundle

• **OBSERVATIONS**:

• (1) since there are 2 MEM units, we can only have **at most** 2 LDs in one bundle, otherwise we have a structural hazard and the bundle could never issue !

• (2) since there are 2 FP ADD units, we can place **at most** 2 FP-ADDs into one bundle , otherwise we have a structural hazard and the bundle could never issue !

• there may be several other good ways to schedule the code for these 2 examples

ITANIUM Scheduling Example

Recall our Basic Loop in a 5 stage pipeline, before multiple-i
Lets use a new timing-table format, with 5 columns

Assumptions:
(1) One FP ADD, 3 stage pipeline (before multiple-issue)
(2) one FP MULT, 6 stage pipeline
(3) 1 Integer ALU unit
(4) 1 MEM unit, 1 stage
(5) Data forwarding happens at the end of the cc in which the result is produced
(6) the EX column shows the execution times in INT ALU unit, and FP units
(7) an Instruction goes into the MEM stage only if it's a LD or SD
otherwise, it bypasses the MEM stage and goes straight to the Write-Back stage

	Instruction	Issue (ID stage)	Execute	Mem	Write-Back	comment
loop	LD F0, 0(R1)	1	2	3	4	fetch X(I)
	ADDD F1, F0, F31	2..3	4..6		7	add scalar ir
	SD F1, 0(R1)	4..5	6	7		
	ADDI R1, R1,#-8	6	7		8	
	BNEZ R1,loop	7	8			
	No-op					

• Suppose we add hardware to support these next 2 assumptions:

• New Assumption #1: Instructions that don't use MEM don't pass through MEM stage(s)

• New Assumption #2: Instructions that don't use WB don't pass through WB stage(s)

Itanium - Basic Loop, Unrolled(4) & Scheduled MAX-SPEED

```
basic loop     LD     FO,0(R1)
               SUBI   R1,R1,#8
               ADDD   F1,F0,F3:
               BNEZ   R1,loop
               SD     F1,+8(R1
```

Assumptions:
(1) 2 FP ADD units, each a 3 stage pipeline
(2) one FP MULT unit, a 6 stage pipeline
(3) 4 Integer ALU units (to avoid structural hazards)
(4) 2 MEM units, each a 2 stage pipeline
(5) Data forwarding happens at the end of the cc in which the result is produced
(6) EX column shows execution times in both INT and FP units
(7) Instruction goes to MEM stage only if it's a LD or SD
(8) Instructions can bypass MEM or WB stages if they are not needed

Unrolled 4 times and Scheduled for MAX SPEED

Slot #1	Slot #2	Slot #3	Issue	Ex(1)	Ex(2)	Ex(3)	M(1)	M(2)	WB(1)	WB(2)	WB(3)	Comment/Hazard
LD F0, 0(R1)	LD F1, -8(R1)	no-op	1	2	2		3..4	3..4	5	5		
LD F2, -16(R1)	LD F3, -24(R1)	no-op	2	3	3		4..5	4..5	6	6		
no-op	ADDD F4,F0,F31	ADDD F5,F1,F31	3,4		5..7	5..7			8	8		wait for F0,F1
SUBI R1,R1,#32	ADDD F6,F2,F31	ADDD F7,F3,F31	5	6	6..8	6..8			7	9	9	bypass M stage
SD F4, +32(R1)	SD F5, +24(R1)	BNEZ R1,loop	6	7	7	7	8..9	8..9				bypass WB stage
SD F6, +16(R1)	SD F7, +8(R1)	no-op	7	8	8		9..10	9..10				bypass WB stage

- 7 cc for 4 iterations: at 2 GHz clock, performance = (2 GHz) * (4 flops/7 cc) = 1,143 GigaFlops/sec

Itanium - Basic Loop, Unrolled(4) & Scheduled MIN-BUNDLES

```
basic loop     LD     FO,0(R1)
               SUBI   R1,R1,#8
               ADDD   F1,F0,F3:
               BNEZ   R1,loop
               SD     F1,+8(R1
```

Slot #1	Slot #2	Slot #3	Issue	Ex(1)	Ex(2)	Ex(3)	Mem(1	Mem(2	WB(1)	Wb(2)	WB(3)	Comments
LD F0, 0(R1)	LD F1, -8(R1)	no-op	1	2	2		3..4	3..4	5	5		
LD **F2**, -16(R1)	LD F3, -24(R1)	ADDD F4,F0,F31	2..4	5	5	5..7	6..7	6..7	**8**	8	8	
SUBI R1,R1,#32	ADDD F5,F1,F31	ADDD F6,**F2**,F31	5..7	8	8..10	8..10			9	11	11	forward F2 end cc 7
SD F4, +32(R1)	SD F5, +24(R1)	ADDD **F7**,F3,F31	8..9	10	10	10..12	11..12	11..12			13	forward F5
SD F6, +16(R1)	SD **F7**, +8(R1)	BNEZ R1,loop	11	12	12	12	13..14	13..14				

- This code uses 5 Issue-Bundles, compared with 6 Issue-Bundles on the previous slide. It executes in 11 cc, compared with 7 cc on the previous slide.

- This is just one possible way to schedule the instructions for maximum code density. In general, using strategy #2 (maximize code density), we really should maximize code density as a first criterion, and and then maximize performance (minimize stalls) when possible.

Real Itanium-1 Latencies & Penalties

(ref text - ch. 4.7, pg. 359)

Instruction	Latency
Integer load	1
Floating-point load	9
Correctly predicted taken branch	0–3
Mispredicted branch	9
Integer ALU operations	0
FP arithmetic	4

- recall that **latency** = # clock cycles that must expire in between 2 dependent instructions, ie an instruction producing data and an instruction consuming the data
- observe the large latencies of **FP loads (9 cc)** and **mis-predicted branches (9 cc)**
- each cc represents 3 instructions**, so a 9 cc penalty** = 27 instruction slots ! These are huge penalties

Itanium-1 Performance (ref text - ch.4 , pg. 360)

- on INT SPEC benchmarks, the Itanium-1 has 60 % of the performance of the Pentium-4
- on FP SPEC benchmarks, the Itanium-1 is about 10 % faster than the Pentium-4, when using a clock rate which is slower than the P4 clock rate
- the Itanium-1 has a **4 MB off-chip L3 cache**, the P4 does not have an L3 cache
- interestingly, the Itanium-1 gets its improved FP performance over the Pentium-4 based on only one program in the benchmark, where it is 4 times faster; if that program would be excluded, the Itanium-1 would be slower than the P4 !
- the Itanium-1 performance appears to be due to the **L3 cache;** a 4MB L3 cache would offer a performance improvement to **any machine, since this is a very large cache**
- in terms of FP power per watt, the Itanium-1 has only 56 % of the performance of the P4 (half as efficient !)

Itanium-2 Performance - Sept 2005 (from www)

IDEAS Top Performers - SPECfp2000 - Single CPU Subset

Rank	Company	System	Processor	Peak Result	Baseline	Test Date
1	IBM Corporation	IBM System p5 520 (1900 MHz, 1 CPU)	POWER5+	3030	2839	Sep-05
2	IBM Corporation	IBM IntelliStation POWER 285 Workstation (1900 MHz, 1 CPU)	POWER5+	3027	2838	Sep-05
3	IBM Corporation	IBM System p5 550 (1900 MHz, 1 CPU)	POWER5+	3007	2815	Sep-05
4	HITACHI	HITACHI BladeSymphony (1.66GHz/9MB Itanium 2)	Intel Itanium 2		2801	Jun-05
5	Hewlett-Packard Company	HP Integrity rx4640-8 (1.6GHz/9MB Itanium 2)	Intel Itanium 2	2712	2712	Oct-04
6	Hewlett-Packard Company	HP Integrity rx1620-2 (1.6GHz/3MB Itanium 2)	Intel Itanium 2 (1.6 GHz/3MB, 533 MHz FSB)	2692	2692	Oct-04
7	Hewlett-Packard Company	HP Integrity rx2620-2 (1.6GHz/6MB, Itanium 2)	Intel Itanium 2 (1.6 GHz/6MB, 400MHz FSB)	2675	2675	Dec-04
8	SGI	SGI Altix 3700 Bx2 (1600MHz 9M L3, Itanium 2)	Intel Itanium 2		2647	Oct-04
9	SGI	SGI Altix 3700 Bx2 (1600MHz 6M L3, Itanium 2)	Intel Itanium 2		2600	Oct-04
10	IBM Corporation	IBM eServer p5 595 (1900 MHz, 1 CPU)	POWER5	2796	2585	Jan-05

- in 2005, the Itanium-2 seems to be nearly as powerful as the IBM Power5+ **dynamically scheduled** machine
- in the future, it will be interesting to see which architecture dominates (static vs dynamic scheduling)

Dual-Core Itanium Performance - 2006 (from www)

JULY 18, 2006

Montecito Arrives, Doubles Itanium Performance

Intel finally unveiled its long-awaited dual-core Montecito chip, and replaced the familiar codename with the unwieldy moniker: "Dual-Core Intel Itanium 2 Processor 9000 Series." The fastest chips run at the same 1.6 GHz as did the single-core Madison 9M top end part – the widely anticipated clock rate, but slower than once had been targeted using the "Foxton" speed booster. Thanks to the two cores per processor chip, Montecito delivers approximately twice the performance compared to its single-core predecessor. To offset contention for the shared front side bus, Intel increased the on-chip L3 cache to 24 MB (shared by both cores) which consume much of the remarkable 1.72 billion transistors on the die, triple the circuit count of its predecessor.

Although Montecito is an impressive implementation, it would have been far more imposing had it shipped in 2005, as once was expected. At this stage, Montecito faces formidable competition from IBM's POWER5+ processors, which currently hold top position on many industry standard benchmarks. Undoubtedly Montecito will wrest some of those benchmarks away from POWER, but look for leapfrogging results as vendors tune and tweak to regain the leadership crowns.

- in 2006, it looks like the dynamically-scheduled IBM Power5+ still is in top spot

Limits on ILP (ref text - section 4.7, pg.322)

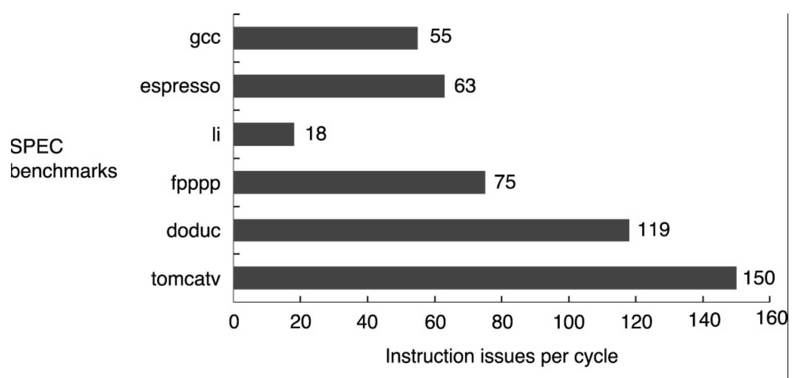
• Consider an ideal superscalar CPU, and a compiler which can look at assembler instructions, make compiler-time branch predictions, determine data-dependencies and schedule instructions to maximize parallelism

• To check the data-dependencies between 50 instructions requires over 2,000 comparisons by the compiler; to check the dependencies between 2000 instructions requires over 4 million comparisons !

• suppose the compiler processes "windows" of instructions looking for parallelism

• what are the limits to ILP, for future machines ?

Limits on ILP (ref text - fig 3.35, pg 242)



• Above table shows how many instructions can issue together after scheduling, for various applications

• In the future, we may have statically scheduled multiple-issue machines which issue between 64 and 128 instructions per cc

Notes