Instruction Level Parallelism and Superscalar Processors

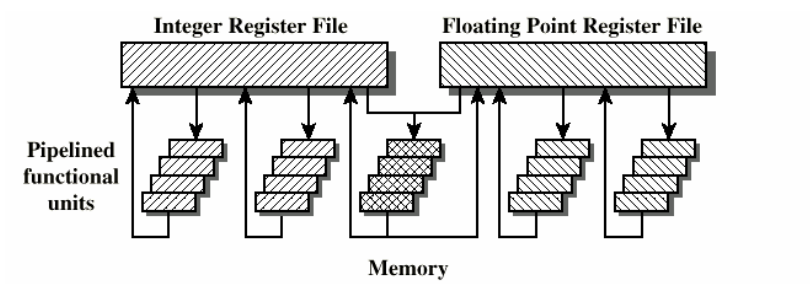
What is Superscalar?

- Common instructions (arithmetic, load/store, conditional branch) can be initiated and executed independently
- Equally applicable to RISC & CISC
- In practice usually RISC

Why Superscalar?

- Most operations are on scalar quantities (see RISC notes)
- Improve these operations to get an overall improvement

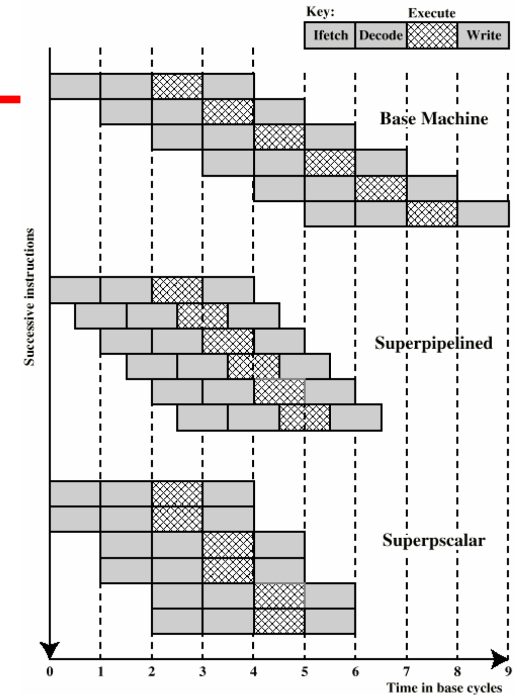
General Superscalar Organization



Superpipelined

- Many pipeline stages need less than half a clock cycle
- Double internal clock speed gets two tasks per external clock cycle
- Superscalar allows parallel fetch execute

Superscalar v Superpipeline



Limitations

- Instruction level parallelism
- Compiler based optimisation
- Hardware techniques
- Limited by
  - —True data dependency
  - —Procedural dependency
  - —Resource conflicts
  - —Output dependency
  - —Antidependency

True Data Dependency

- ADD r1, r2 (r1 := r1+r2;)
- MOVE r3,r1 (r3 := r1;)
- Can fetch and decode second instruction in parallel with first
- Can NOT execute second instruction until first is finished

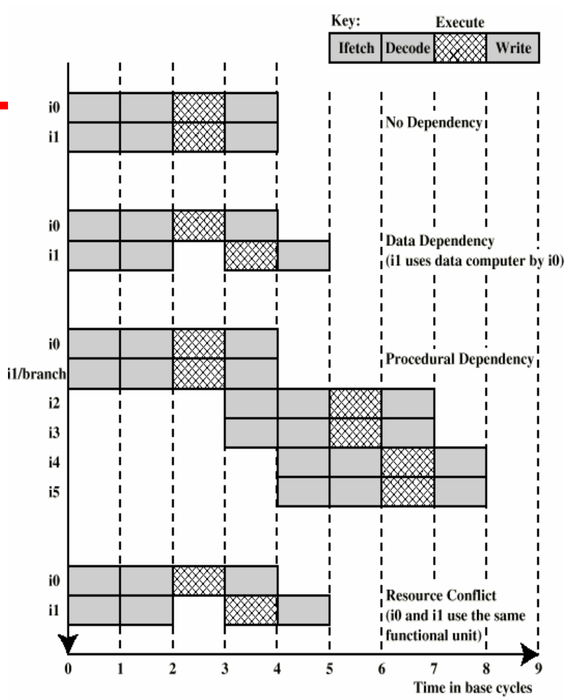
Procedural Dependency

- Can not execute instructions after a branch in parallel with instructions before a branch
- Also, if instruction length is not fixed, instructions have to be decoded to find out how many fetches are needed
- This prevents simultaneous fetches

Resource Conflict

- Two or more instructions requiring access to the same resource at the same time
  - e.g. two arithmetic instructions
- Can duplicate resources
  - e.g. have two arithmetic units

Effect of Dependencies



Design Issues

- Instruction level parallelism
  - Instructions in a sequence are independent
  - Execution can be overlapped
  - Governed by data and procedural dependency
- Machine Parallelism
  - Ability to take advantage of instruction level parallelism
  - Governed by number of parallel pipelines

Instruction Issue Policy

- Order in which instructions are fetched
- Order in which instructions are executed
- Order in which instructions change registers and memory

In-Order Issue In-Order Completion

- Issue instructions in the order they occur
- Not very efficient
- May fetch >1 instruction
- Instructions must stall if necessary

In-Order Issue In-Order Completion (Diagram)

Decode		Execute			Write		Cycle
I1	I2						1
I3	I4	I1	I2				2
I3	I4	I1					3
	I4			I3	I1	I2	4
I5	I6			I4			5
	I6		I5		I3	I4	6
			I6				7
					I5	I6	8

In-Order Issue Out-of-Order Completion

- Output dependency
  - —R3:= R3 + R5; (I1)
  - —R4:= R3 + 1; (I2)
  - —R3:= R5 + 1; (I3)
  - —I2 depends on result of I1 - data dependency
  - —If I3 completes before I1, the result from I1 will be wrong - output (read-write) dependency

In-Order Issue Out-of-Order Completion (Diagram)

Decode		Execute			Write		Cycle
I1	I2						1
I3	I4	I1	I2				2
	I4	I1		I3	I2		3
I5	I6			I4	I1	I3	4
	I6		I5		I4		5
			I6		I5		6
					I6		7

Out-of-Order Issue Out-of-Order Completion

- Decouple decode pipeline from execution pipeline
- Can continue to fetch and decode until this pipeline is full
- When a functional unit becomes available an instruction can be executed
- Since instructions have been decoded, processor can look ahead

Out-of-Order Issue Out-of-Order Completion (Diagram)

Decode		Window	Execute			Write		Cycle
I1	I2							1
I3	I4	*I1,I2*	I1	I2				2
I5	I6	*I3,I4*	I1		I3	I2		3
		I4,I5,I6		I6	I4	I1	I3	4
		I5		I5		I4	I6	5
						I5		6

Antidependency

- Write-write dependency
  - —R3:=R3 + R5; (I1)
  - —R4:=R3 + 1; (I2)
  - —R3:=R5 + 1; (I3)
  - —R7:=R3 + R4; (I4)
  - —I3 can not complete before I2 starts as I2 needs a value in R3 and I3 changes R3

Register Renaming

- Output and antidependencies occur because register contents may not reflect the correct ordering from the program
- May result in a pipeline stall
- Registers allocated dynamically
  - —i.e. registers are not specifically named

Register Renaming example

- R3b:=R3a + R5a (I1)
- R4b:=R3b + 1 (I2)
- R3c:=R5a + 1 (I3)
- R7b:=R3c + R4b (I4)
- Without subscript refers to logical register in instruction
- With subscript is hardware register allocated
- Note R3a R3b R3c

Machine Parallelism

- Duplication of Resources
- Out of order issue
- Renaming
- Not worth duplication functions without register renaming
- Need instruction window large enough (more than 8)

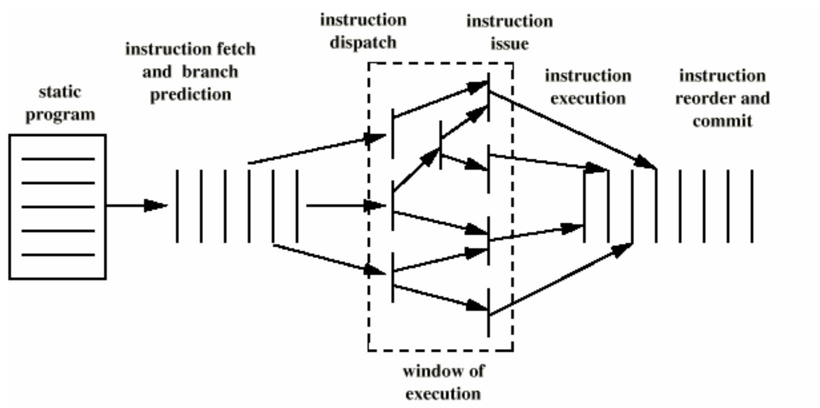
Branch Prediction

- 80486 fetches both next sequential instruction after branch and branch target instruction
- Gives two cycle delay if branch taken

RISC - Delayed Branch

- Calculate result of branch before unusable instructions pre-fetched
- Always execute single instruction immediately following branch
- Keeps pipeline full while fetching new instruction stream
- Not as good for superscalar
  - —Multiple instructions need to execute in delay slot
  - —Instruction dependence problems
- Revert to branch prediction

Superscalar Execution



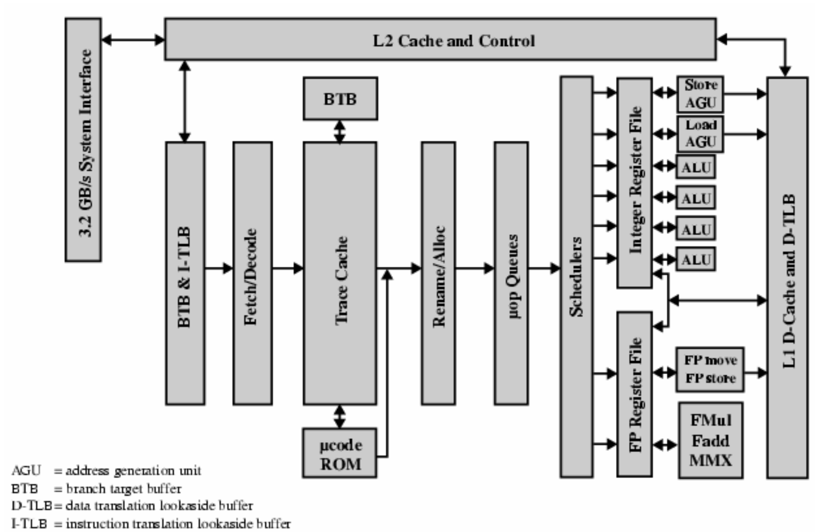
Superscalar Implementation

- Simultaneously fetch multiple instructions
- Logic to determine true dependencies involving register values
- Mechanisms to communicate these values
- Mechanisms to initiate multiple instructions in parallel
- Resources for parallel execution of multiple instructions
- Mechanisms for committing process state in correct order

Pentium 4

- 80486 - CISC
- Pentium – some superscalar components
  - —Two separate integer execution units
- Pentium Pro – Full blown superscalar
- Subsequent models refine & enhance superscalar design

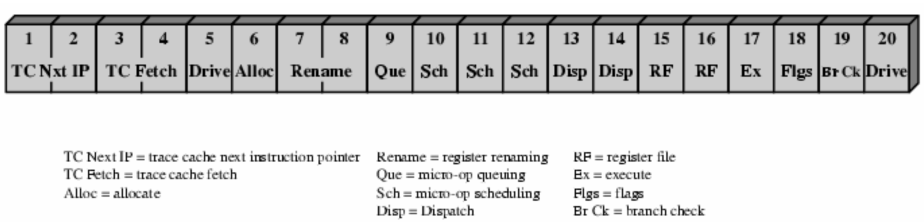
Pentium 4 Block Diagram



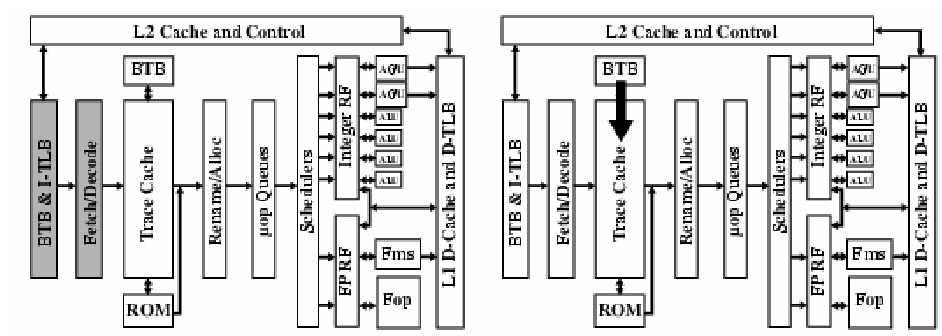
Pentium 4 Operation

- Fetch instructions form memory in order of static program
- Translate instruction into one or more fixed length RISC instructions (micro-operations)
- Execute micro-ops on superscalar pipeline
  - —micro-ops may be executed out of order
- Commit results of micro-ops to register set in original program flow order
- Outer CISC shell with inner RISC core
- Inner RISC core pipeline at least 20 stages
  - —Some micro-ops require multiple execution stages
    - – Longer pipeline
  - —c.f. five stage pipeline on x86 up to Pentium

Pentium 4 Pipeline



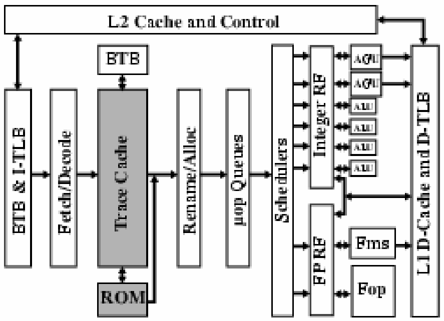
Pentium 4 Pipeline Operation (1)



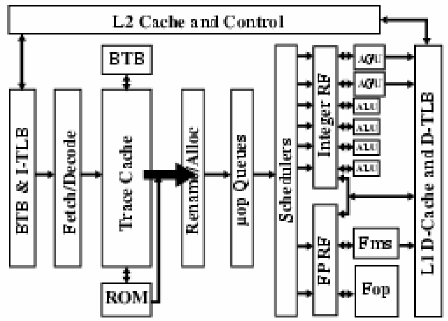
(a) Generation of micro-ops

(b) Trace cache next instruction pointer

Pentium 4 Pipeline Operation (2)

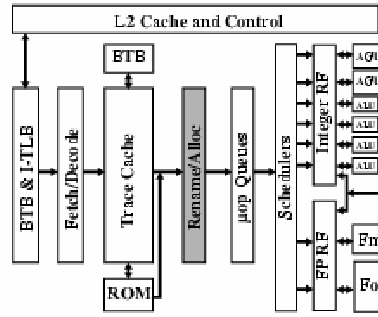


(c) Trace cache fetch

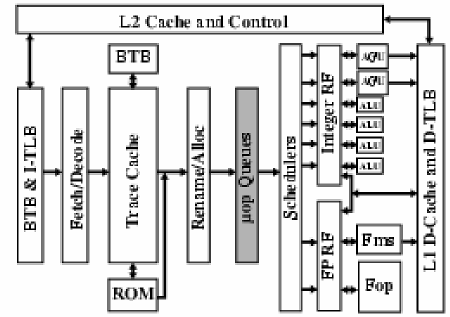


(d) Drive

Pentium 4 Pipeline Operation (3)

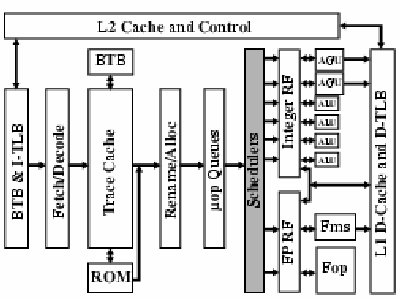


(e) Allocate; Register renaming

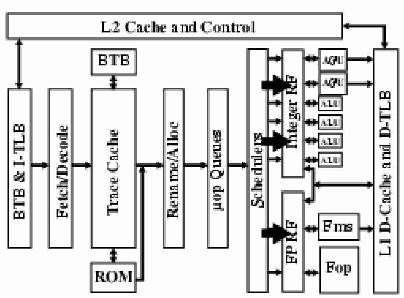


(f) Micro-op queuing

Pentium 4 Pipeline Operation (4)

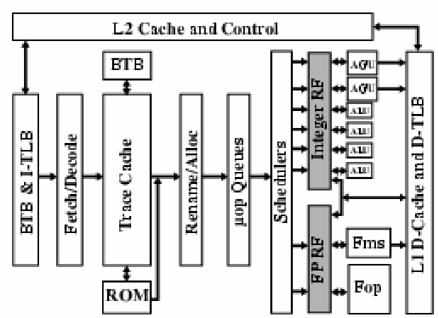


(g) Micro-op scheduling

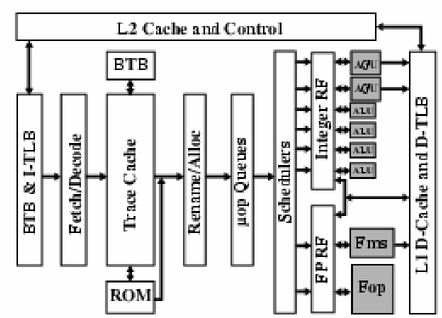


(h) Dispatch

Pentium 4 Pipeline Operation (5)

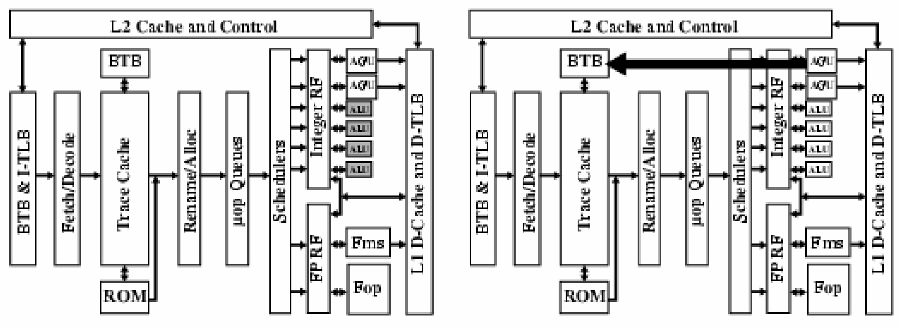


(i) Register file



(j) Execute; flags

Pentium 4 Pipeline Operation (6)

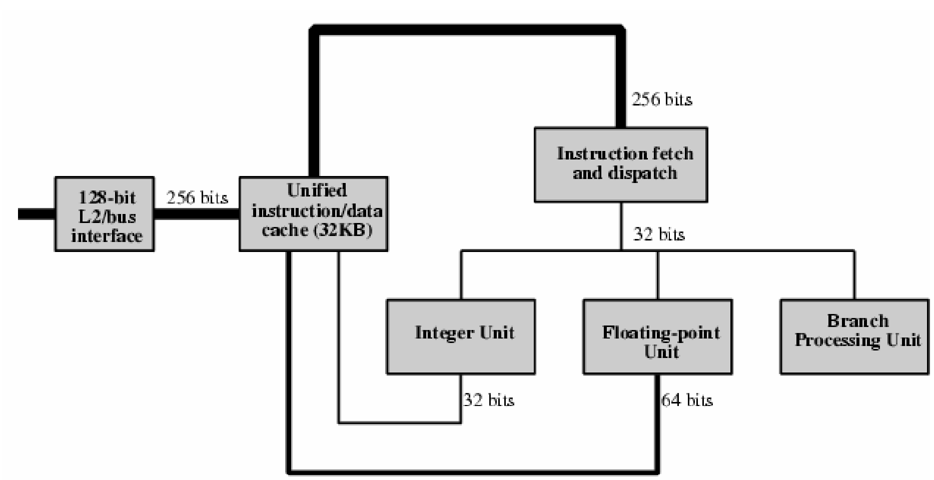


(k) Branch check (l) Branch check result

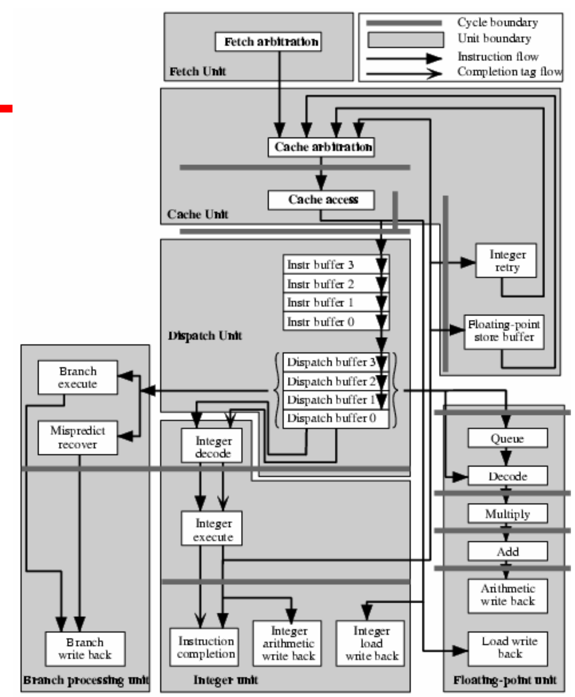
PowerPC

- Direct descendent of IBM 801, RT PC and RS/6000
- All are RISC
- RS/6000 first superscalar
- PowerPC 601 superscalar design similar to RS/6000
- Later versions extend superscalar concept

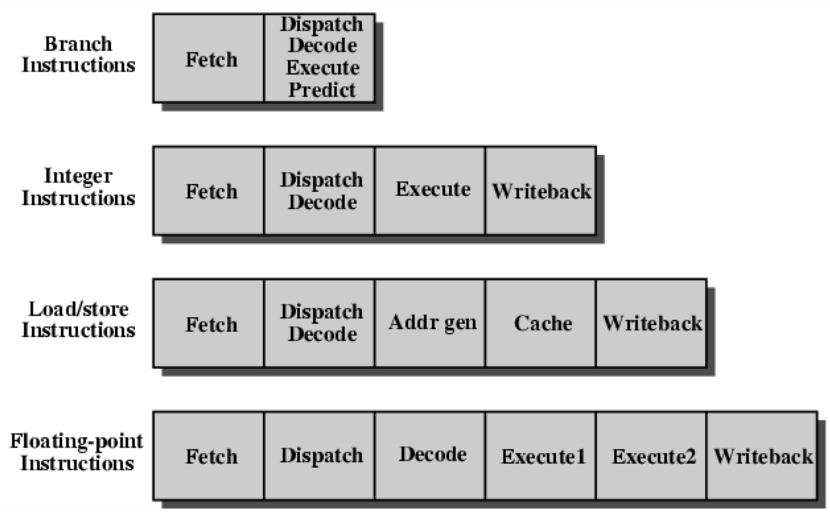
PowerPC 601 General View



PowerPC 601 Pipeline Structure



PowerPC 601 Pipeline



Intel IA-64 Architecture

ITANIUM

Background to IA-64

- Pentium 4 appears to be last in x86 line
- Intel & Hewlett-Packard (HP) jointly developed
- New architecture
  - —64 bit architecture
  - —Not extension of x86
  - —Not adaptation of HP 64bit RISC architecture
- Exploits vast circuitry and high speeds
- Systematic use of parallelism
- Departure from superscalar

Motivation

- Instruction level parallelism
  - —Implicit in machine instruction
  - —Not determined at run time by processor
- Long or very long instruction words (LIW/VLIW)
- Branch predication (not the same as branch prediction)
- Speculative loading
- Intel & HP call this Explicit Parallel Instruction Computing (EPIC)
- IA-64 is an instruction set architecture intended for implementation on EPIC
- Itanium is first Intel product

Superscalar v IA-64

Superscalar	IA-64
RISC-line instructions, one per word	RISC-line instructions bundled into groups of three
Multiple parallel execution units	Multiple parallel execution units
Reorders and optimizes instruction stream at run time	Reorders and optimizes instruction stream at compile time
Branch prediction with speculative execution of one path	Speculative execution along both paths of a branch
Loads data from memory only when needed, and tries to find the data in the caches first	Speculatively loads data before its needed, and still tries to find data in the caches first

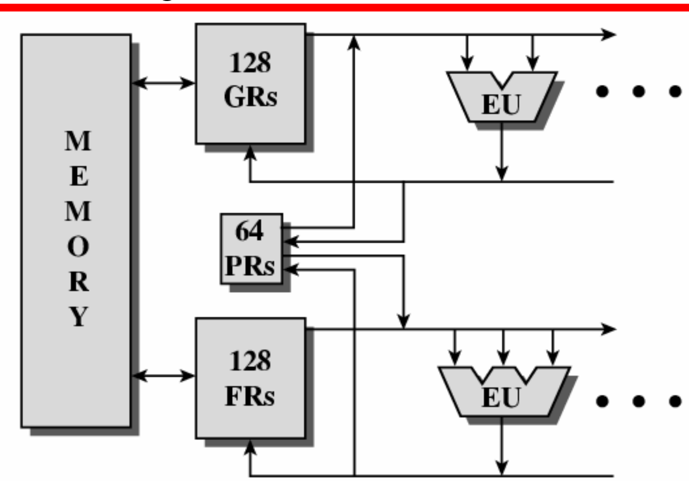
Why New Architecture?

- Not hardware compatible with x86
- Now have tens of millions of transistors available on chip
- Could build bigger cache
  - Diminishing returns
- Add more execution units
  - Increase superscaling
  - "Complexity wall"
  - More units makes processor "wider"
  - More logic needed to orchestrate
  - Improved branch prediction required
  - Longer pipelines required
  - Greater penalty for misprediction
  - Larger number of renaming registers required
  - At most six instructions per cycle

Explicit Parallelism

- Instruction parallelism scheduled at compile time
  - Included with machine instruction
- Processor uses this info to perform parallel execution
- Requires less complex circuitry
- Compiler has much more time to determine possible parallel operations
- Compiler sees whole program

General Organization



GR = General-purpose or integer register
FR = Floating-point or graphics register
PR = One-bit predicate register
EU = Execution unit

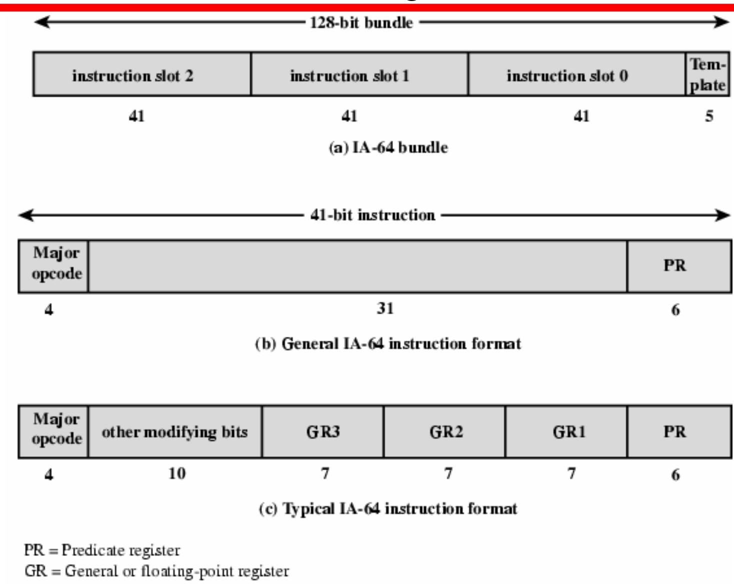
Key Features

- Large number of registers
  - —IA-64 instruction format assumes 256
    - – 128 * 64 bit integer, logical & general purpose
    - – 128 * 82 bit floating point and graphic
  - —64 * 1 bit predicated execution registers (see later)
  - —To support high degree of parallelism
- Multiple execution units
  - —Expected to be 8 or more
  - —Depends on number of transistors available
  - —Execution of parallel instructions depends on hardware available
    - – 8 parallel instructions may be spilt into two lots of four if only four execution units are available

IA-64 Execution Units

- I-Unit
  - —Integer arithmetic
  - —Shift and add
  - —Logical
  - —Compare
  - —Integer multimedia ops
- M-Unit
  - —Load and store
    - – Between register and memory
  - —Some integer ALU
- B-Unit
  - —Branch instructions
- F-Unit
  - —Floating point instructions

Instruction Format Diagram



Instruction Format

- 128 bit bundle
  - —Holds three instructions (syllables) plus template
  - —Can fetch one or more bundles at a time
  - —Template contains info on which instructions can be executed in parallel
    - – Not confined to single bundle
    - – e.g. a stream of 8 instructions may be executed in parallel
    - – Compiler will have re-ordered instructions to form contiguous bundles
    - – Can mix dependent and independent instructions in same bundle
  - —Instruction is 41 bit long
    - – More registers than usual RISC
    - – Predicated execution registers (see later)

Assembly Language Format

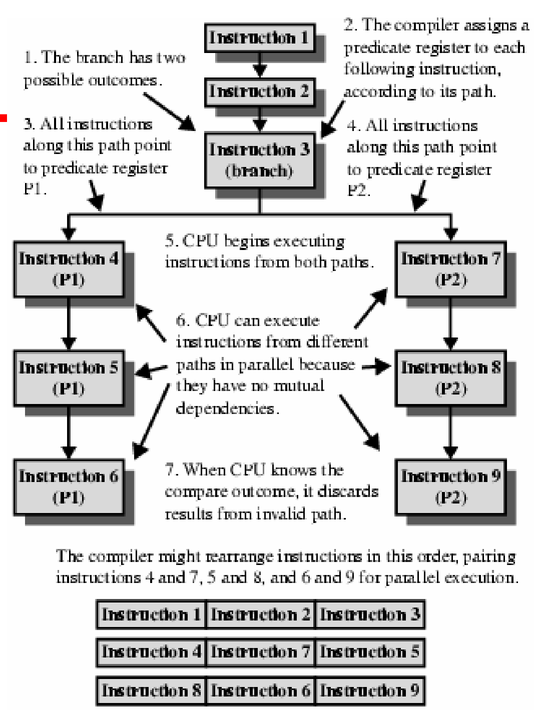
- `[qp] mnemonic [.comp] dest = srcs //`
- qp - predicate register
  - 1 at execution then execute and commit result to hardware
  - 0 result is discarded
- `mnemonic` - name of instruction
- `comp` – one or more instruction completers used to qualify mnemonic
- `dest` – one or more destination operands
- `srcs` – one or more source operands
- `//` - comment
- Instruction groups and stops indicated by `;;`
  - Sequence without read after write or write after write
  - Do not need hardware register dependency checks

Assembly Examples

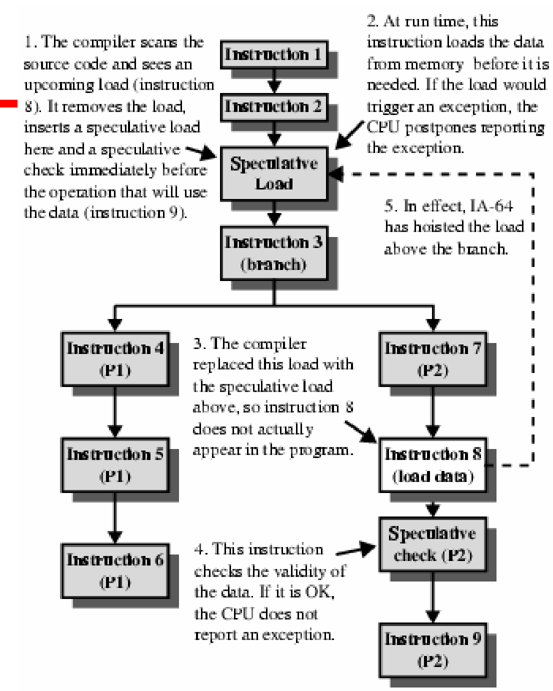
```
ld8 r1 = [r5] ;; //first group
add r3 = r1, r4 //second group
```

- Second instruction depends on value in r1
  - Changed by first instruction
  - Can not be in same group for parallel execution

Predication



Speculative Loading



Control & Data Speculation

- Control
  - —AKA Speculative loading
  - —Load data from memory before needed
- Data
  - —Load moved before store that might alter memory location
  - —Subsequent check in value

Software Pipelining

```
L1:    ld4 r4=[r5],4 ;;   //cycle 0 load postinc 4
       add r7=r4,r9 ;;    //cycle 2
       st4 [r6]=r7,4      //cycle 3 store postinc 4
       br.cloop L1 ;;     //cycle 3
```

- Adds constant to one vector and stores result in another
- No opportunity for instruction level parallelism
- Instruction in iteration *x* all executed before iteration *x+1* begins
- If no address conflicts between loads and stores can move independent instructions from loop *x+1* to loop *x*

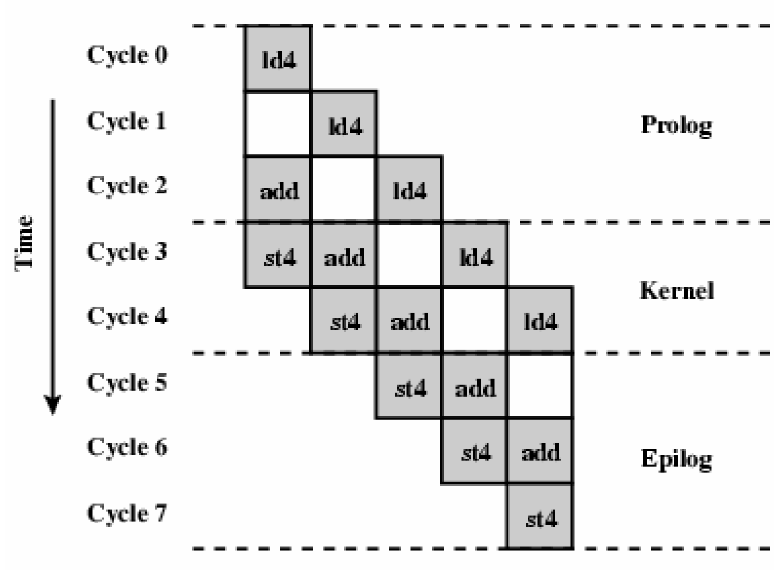
Unrolled Loop

```
ld4 r32=[r5],4;; //cycle 0
ld4 r33=[r5],4;; //cycle 1
ld4 r34=[r5],4   //cycle 2
add r36=r32,r9;; //cycle 2
ld4 r35=[r5],4   //cycle 3
add r37=r33,r9   //cycle 3
st4 [r6]=r36,4;; //cycle 3
ld4 r36=[r5],4   //cycle 3
add r38=r34,r9   //cycle 4
st4 [r6]=r37,4;; //cycle 4
add r39=r35,r9   //cycle 5
st4 [r6]=r38,4;; //cycle 5
add r40=r36,r9   //cycle 6
st4 [r6]=r39,4;; //cycle 6
st4 [r6]=r40,4;; //cycle 7
```

Unrolled Loop Detail

- Completes 5 iterations in 7 cycles
  - —Compared with 20 cycles in original code
- Assumes two memory ports
  - —Load and store can be done in parallel

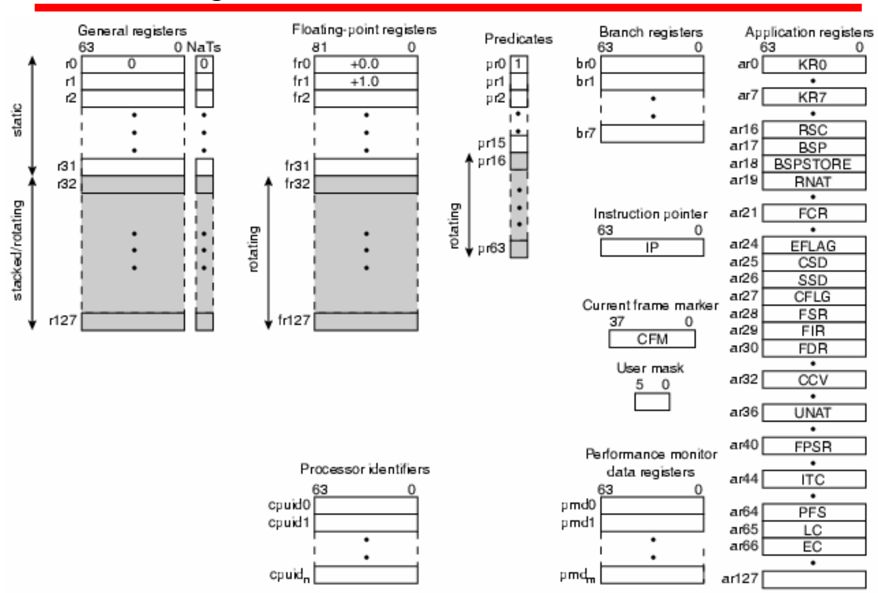
Software Pipeline Example Diagram



Support For Software Pipelining

- Automatic register renaming
  - Fixed size are of predicate and fp register file (p16-P32, fr32-fr127) and programmable size area of gp register file (max r32-r127) capable of rotation
  - Loop using r32 on first iteration automatically uses r33 on second
- Predication
  - Each instruction in loop predicated on rotating predicate register
    - Determines whether pipeline is in prolog, kernel or epilog
- Special loop termination instructions
  - Branch instructions that cause registers to rotate and loop counter to decrement

IA-64 Register Set



IA-64 Registers (1)

- General Registers
  - 128 gp 64 bit registers
  - r0-r31 static
    - references interpreted literally
  - r32-r127 can be used as rotating registers for software pipeline or register stack
    - References are virtual
    - Hardware may rename dynamically
- Floating Point Registers
  - 128 fp 82 bit registers
  - Will hold IEEE 745 double extended format
  - fr0-fr31 static, fr32-fr127 can be rotated for pipeline
- Predicate registers
  - 64 1 bit registers used as predicates
  - pr0 always 1 to allow unpredicated instructions
  - pr1-pr15 static, pr16-pr63 can be rotated

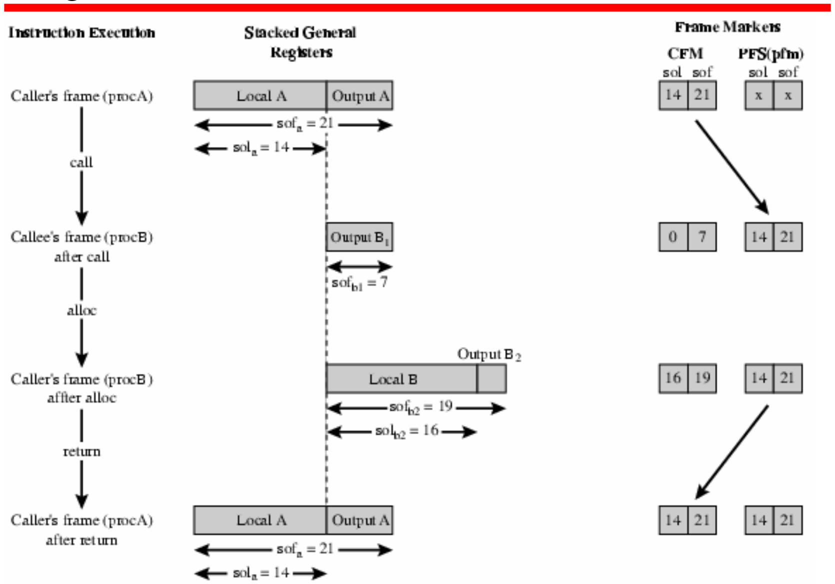
IA-64 Registers (2)

- Branch registers
  - 8 64 bit registers
- Instruction pointer
  - Bundle address of currently executing instruction
- Current frame marker
  - State info relating to current general register stack frame
  - Rotation info for fr and pr
  - User mask
    - Set of single bit values
    - Allignment traps, performance monitors, fp register usage monitoring
- Performance monitoring data registers
  - Support performance monitoring hardware
- Application registers
  - Special purpose registers

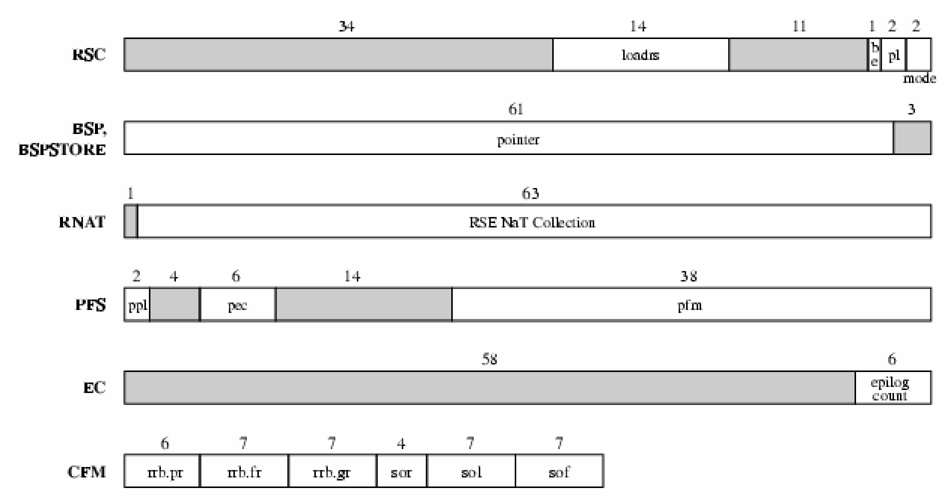
Register Stack

- Avoids unnecessary movement of data at procedure call & return
- Provides procedure with new frame up to 96 registers on entry
  - r32-r127
- Compiler specifies required number
  - Local
  - output
- Registers renamed so local registers from previous frame hidden
- Output registers from calling procedure now have numbers starting r32
- Physical registers r32-r127 allocated in circular buffer to virtual registers
- Hardware moves register contents between registers and memory if more registers needed

Register Stack Behaviour



Register Formats



Itanium Organization

- Superscalar features
  - —Six wide, ten stage deep hardware pipeline
  - —Dynamic prefetch
  - —branch prediction
  - —register scoreboard to optimise for compile time nondeterminism
- EPIC features
  - —Hardware support for predicated execution
  - —Control and data speculation
  - —Software pipelining

Itanium Processor Diagram