CS184b: Computer Architecture [Single Threaded Architecture: abstractions, quantification, and optimizations]

Day11: February 8, 2000

EPIC

Today

- EPIC: next generation of VLIW
- adding recent features/lessons to VLIW
- other modern features…

- …not much quantitative data :-(

VLIW

- Exploit ILP
- w/out all the hardware complexity and cost
- Relegate even more interesting stuff to the compiler (REMISC?)
- …but no binary compatibility path

Scaling Idea

- **Problem:**
  - VLIW: amount of parallelism fixed by VLIW schedule
  - SuperScalar: have to check many dynamic dependencies
- **Idealized Solution:**
  - expose all the parallelism you can
  - run it as sequential/parallel as necessary

Basic Idea

- What if we scheduled an *infinitely* wide VLIW?
- For an N-issue machine
  - for I = 1 to (width of this instruction/N)
    - grab next N instructions and issue

Problems?

- Instructions arbitrarily long?
- Need infinite registers to support infinite parallelism?
- Split Register file still work?
- Sequentializing semantically parallel operations introduce hazards?

Instruction Length

- Field in standard way
  - *pinsts* (from cs184a)
  - like RISC instruction components
- Allow variable fields (syllables) per parallel component
- Encode
  - stop bit (break between instructions)
  - (could have been length…)

Registers

- Compromise on fixed number of registers
  - …will limit parallelism, and hence scalability…
- Also keep(adopt) monolithic/global register file
  - syllables can't control which "cluster" in which they'll run
  - consider series of 7 syllable ops
    - where syllables end up on 3-issue, 4-issue machine?

Sequentializing Parallel

- Consider wide instruction:
  - `MUL R1,R2,R3  ADD R2,R1,R5`
- Now sequentialize:
  - `MUL R1,R2,R3`
  - `ADD R2,R1,R5`
- Different semantics

Semantics of a "Long Instruction"

- Correct if executed in parallel
- Preserved with sequentialization
- So:
  - read values are from beginning of issue
  - no RAW hazards:
    - can't write to a register used as a source
  - no WAW hazards:
    - can't write to a register multiple times

Non-VLIW-ness

Register File

- Monolithic register file
- Ports grows with number of physical syllables supported

Instruction Issue

- VLIW
  - no interconnect/shuffle/etc. between fetch and clusters
  - straight (like fixed fields)
- EPIC
  - variable length and varying syllable support
  - ~~may have to perform arbitrary shift from memory fetch to syllable issue?~~
    - ~~$O(N^2)$ wire interconnect?~~
    - ~~…but don't have to compare values to control...~~

Believe can see away to keep it local

Bypass

- VLIW
  - schedule around delay cycles in pipe
- EPIC not know which instructions in pipe at compile time
  - do have to watch for hazards between instruction groups
  - ? Similar pipelining issues to RISC/superscalar?
  - Bypass only at issue group boundary
    - maybe can afford to be more spartan?

Concrete Details

(IA-64)

Terminology

- Syllables (their *pinsts*)
- bundles: group of 3 syllables for IA-64
- Instruction group: “variable length” issue set
  - *i.e.* set of bundles (syllables) which may execute in parallel

IA-64 Encoding

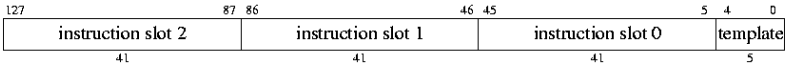


Figure 3-16. Bundle Format

Instruction Type	Description	Execution Unit Type
A	Integer ALU	I-unit or M-unit
I	Non-ALU integer	I-unit
M	Memory	M-unit
F	Floating-point	F-unit
B	Branch	B-unit
L+X	Extended	I-unit

Source: Intel/HP IA-64 Application ISA Guide 1.0

IA-64 Templates

Table 3-9. Template Field Encoding and Instruction Slot Mapping

Template	Slot 0	Slot 1	Slot 2
00	M-unit	I-unit	I-unit
01	M-unit	I-unit	I-unit
02	M-unit	I-unit	I-unit
03	M-unit	I-unit	I-unit
04	M-unit	L-unit	X-unit
05	M-unit	L-unit	X-unit
06			
07			
08	M-unit	M-unit	I-unit
09	M-unit	M-unit	I-unit
0A	M-unit	M-unit	I-unit
0B	M-unit	M-unit	I-unit
0C	M-unit	F-unit	I-unit
0D	M-unit	F-unit	I-unit
0E	M-unit	M-unit	F-unit
0F	M-unit	M-unit	F-unit
10	M-unit	I-unit	B-unit
11	M-unit	I-unit	B-unit
12	M-unit	B-unit	B-unit
13	M-unit	B-unit	B-unit
14			
15			
16	B-unit	B-unit	B-unit
17	B-unit	B-unit	B-unit
18	M-unit	M-unit	B-unit
19	M-unit	M-unit	B-unit
1A			
1B			
1C	M-unit	F-unit	B-unit
1D	M-unit	F-unit	B-unit
1E			
1F			

Instruction Type	Description	Execution Unit Type
A	Integer ALU	I-unit or M-unit
I	Non-ALU integer	I-unit
M	Memory	M-unit
F	Floating-point	F-unit
B	Branch	B-unit
L+X	Extended	I-unit

Source: Intel/HP IA-64 Application ISA Guide 1.0

IA-64 Registers

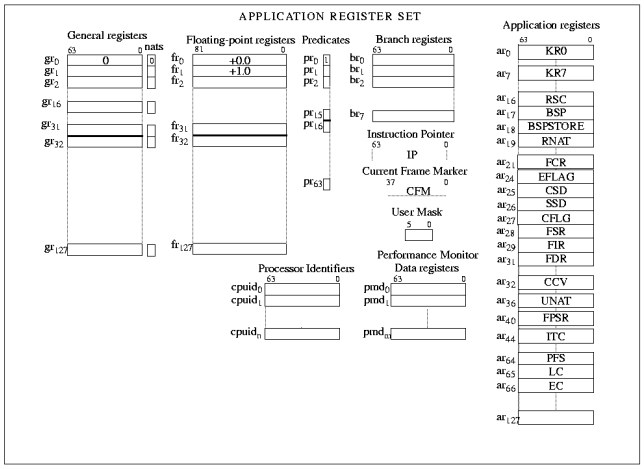


Figure 3-1. Application Register Model

Source: Intel/HP IA-64 Application ISA Guide 1.0

Other Additions

Other Stuff

- Speculation/Exceptions
- Predication
- Branching
- Memory
- Register Renaming

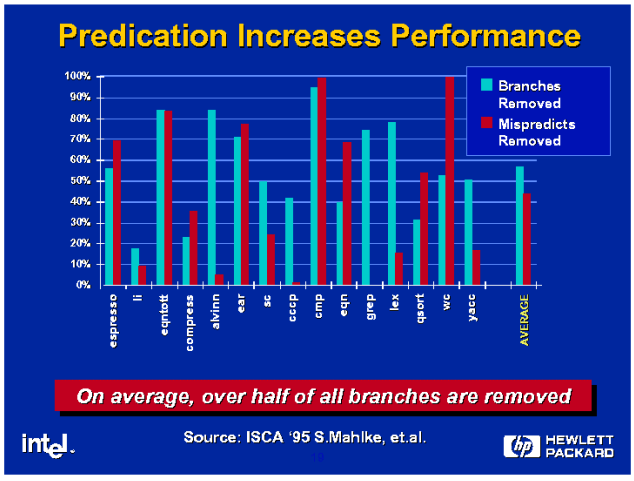
Speculation

- Can mark instructions as speculative
- Bogus results turn into designated NaT
  - particularly loads
  - compare posion bits
- NaT arithmetic produces NaTs
- Check for NaTs if/when care about result

Predication

- Already seen conditional moves
- Almost every operation here is conditional
  - (similar to ARM?)
- Full set of predicate registers
  - few instructions for calculating composite predicates
- Again, exploit parallelism and avoid loosing trace on small, unpredictable branches
  - can be better to do both than branch wrong

Predication: Quantification



Branching

- Unpack branch
  - branch prepare (calculate target)
    - added branch registers for
  - compare (will I branch?)
  - branch execute (transfer control now)
- sequential semantics w/in instruction group
- indicate static or dynamic branch predict
- loop instruction (fixed trip loops)
- multiway branch (with predicates)

Memory

- Prefetch
  - typically non-binding?
- control caching
  - can specify not to allocate in cache
    - if know use once
    - suspect no temporal locality
  - can specify appropriate cache level
- speculation

Memory Speculation

- Ordering limits due to aliasing
  - don't know if can reorder a[i], a[j]
    - A[j]=x+y;
    - C=a[i]*Z;
  - might get WAR hazards
- Memory speculation:
  - reorder read
  - check in order and correct if incorrect

Memory Speculation

- Store(st_addr,data)
- load(ld_addr,target)
- use(target)

- Aload(ld_addr,target)
- store(st_adder,data)
- acheck(target,recovery_addr)
- use(target)

Memory Speculation

Before Data Speculation	After Data Speculation

`// other instructions`
`st8 [r4] = r12`
`ld8 r6 = [r8];;`
`add r5 = r6, r7;;`
`st8 [r18] = r5`
	`ld8.a r6 = [r8];; // advanced load`
	`// other instructions`
	`st8 [r4] = r12`
	`ld8.c.clr r6 = [r8] // check load`
	`add r5 = r6, r7;;`
	`st8 [r18] = r5`

Figure 4-2. Data Speculation Recovery Using ld.c

If advanced load fails, checking load performs actual load.

Memory Speculation

Before Data Speculation	After Data Speculation

`// other instructions`
`st8 [r4] = r12`
`ld8 r6 = [r8];;`
`add r5 = r6, r7;;`
`st8 [r18] = r5`
	`ld8.a r6 = [r8];;`
	`// other instructions`
	`add r5 = r6, r7;;`
	`// other instructions`
	`st8 [r4] = r12`
	`chk.a.clr r6, recover`
	`back:`
	`st8 [r18] = r5`
	
	`// somewhere else in program`
	`recover:`
	`ld8 r6 = [r8];;`
	`add r5 = r6, r7`
	`br back`

Figure 4-3. Data Speculation Recovery Using chk.a

If advanced load succeeds, values are good and can continue; otherwise have to execute patch up code.

Advanced Load Support

- Advanced Load Table
- Speculative loads allocate space in ALAT
  - tagged by target register
- ALAT checked against stores
  - invalidated if see overwrite
- At check or load
  - if find valid entry, advanced load succeeded
  - if not find entry, failed
    - reload …or…
    - branch to patchup code

Register "renaming"

- Use top 96 registers like a stack?
- Still register addressable
- But increment base on
  - loops, procedure entry
- Treated like stack with "automatic" background task to save/restore values

Register "renaming"

- Application benefits:
  - software pipelining without unrolling
  - values from previous iterations of loop get different names (rename all registers allocated in loop by incrementing base)
    - allows reference to by different names
  - pass data through registers
    - without compiling caller/callee together
    - variable number of registers

Register "Renaming"

- …old bad idea?
  - Stack machines?
    - Does allow register named access
  - Register Windows (RISC-II,SPARC)
    - SPARC register windows were fixed size
    - had to save and restore in that sized chunk
    - only window-set visible

Register “renaming” Costs

- Slow down register access
  - have to do arithmetic on register numbers
- Require hardware register save/restore engine
  - orthogonal task to execution
  - complicated?
- Complicates architecture

Admin

- Tuesday:
  - No CLASS
- Thursday:
  - start on caching
  - (so start reading chapter 5)

Big Ideas

- Compile for maximum parallelism
- Sequentialize as necessary
  - (moderately) cheap

Big Ideas

- Latency reduction hard
  - path length is our parallelism limiter
  - often good to trade more work for shorter critical path
    - area-time tradeoff
  - speculation, predication reduce path length
    - perhaps at cost of more total operations

Big Ideas [MSB-1]

- Local control (predication)
  - costs issue
  - increases predictability, parallelism
- Common Case/Speculation
  - avoid worst-case pessimism on memory operations
  - common case faster
  - correct in all cases