

A (naïve) Primer on VLIW – EPIC with lots of slides borrowed/edited from an Intel-HP presentation

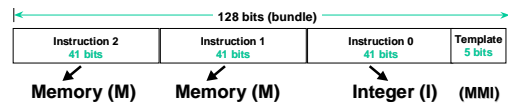
- VLIW direct descendant of horizontal microprogramming
 - Two commercially unsuccessful machines: Multiflow and Cydrome
- Compiler generates instructions that can execute together
 - Instructions executed in order and assumed to have a fixed latency
- Difficulties occur with :
 - Branch prediction -> Use of predication
 - Pointer-based computations -> Use cache hints and speculative loads
 - Unpredictable latencies (e.g., cache misses)

11/2/01

VLIW CSE 471 Autumn 01

1

IA-64 : Explicitly Parallel Architecture



- IA-64 template specifies
 - The type of operation for each instruction, e.g.
 - MFI, MMI, MII, MLI, MIB, MMF, MFB, MMB, MBB, BBB
 - Intra-bundle relationship, e.g.
 - M / MI or MI / I (/ is a "stop" meaning no parallelism)
 - Inter-bundle relationship
- Most common combinations covered by templates
 - Headroom for additional templates
- Simplifies hardware requirements
- Scales compatibly to future generations

M=Memory
F=Floating-point
I=Integer
L=Long Immediate
B=Branch

11/2/01

VLIW CSE 471 Autumn 01

2

(Merced) Itanium implementation

- Can execute 2 bundles (6 instructions) per cycle
- 10 stage pipeline
- 4 integer units (2 of them can handle load-store), 2 f-p units and 3 branch units
- Issue in order, execute in order but can complete out of order. Uses a (restricted) register scoreboard technique to resolve dependencies.

11/2/01

VLIW CSE 471 Autumn 01

3

Itanium implementation (?)

- **Predication** reduces number of branches and number of mispredicts,
- Nonetheless: sophisticated branch predictor
 - Compiler hints: BPR instruction provides "easy" to predict branch address and addresses; reduces number of entries in BTB
 - Two-level hardware prediction *Sas* (4,2) (512 entry local history table 4-way set-associative, indexing 128 PHT –one per set- each with 16 entries –2-bit saturating counters). Number of bubbles on predicted branch taken: 2 or 3
 - And a 64-entry BTB (only 1 cycle stall)
 - Mispredicted branch penalty: 9 cycles

11/2/01

VLIW CSE 471 Autumn 01

4

Itanium implementation (?)

- There are "instruction queues" between the fetch unit and the execution units. Therefore branch bubbles can often be absorbed because of long latencies (and stalls) in the execute stages

11/2/01

VLIW CSE 471 Autumn 01

5

IA-64 for High Performance

- Number of branches in large server apps overwhelm traditional processors
 - IA-64 **predication** removes branches, avoids mispredicts
- Environments with a large number of users require high performance
 - IA-64 uses **speculation to reduce impact of memory latency**
 - 64-bit addressing enables systems with very large virtual and physical memory

11/2/01

VLIW CSE 471 Autumn 01

6

Middle Tier Application Needs

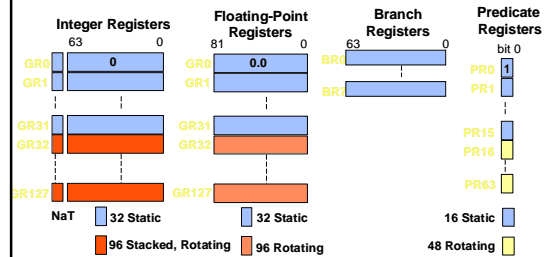
- Mid-tier applications have diverse code requirements
 - Integer code with many small loops
 - Significant call / return requirements (C++, Java)
- IA-64's register model supports these various requirements
 - Large register file provides significant resources for optimized performance
 - Rotating registers enables efficient loop execution
 - Register stack to handle call-intensive code

11/2/01

VLIW CSE 471 Autumn 01

7

IA-64's Large Register File



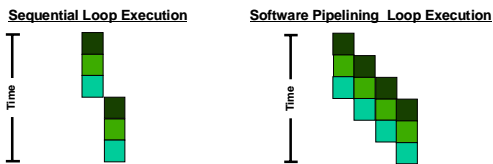
11/2/01

VLIW CSE 471 Autumn 01

8

Software Pipelining via Rotating Registers

As seen before



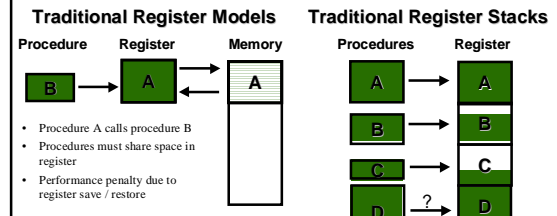
- Traditional architectures need complex software loop unrolling for pipelining
 - Results in code expansion --> Increases cache misses --> Reduces performance
- IA-64 utilizes rotating registers to achieve software pipelining
 - Avoids code expansion --> Reduces cache misses --> Higher performance

11/2/01

VLIW CSE 471 Autumn 01

9

Traditional Register Models



- Procedure A calls procedure B
- Procedures must share space in register
- Performance penalty due to register save / restore

I think that the "traditional register stack" model they refer to is the "register windows" model used in Sparc

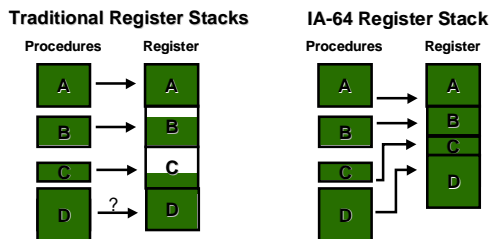
- Eliminate the need for save / restore by reserving fixed blocks in register
- However, fixed blocks waste resources

11/2/01

VLIW CSE 471 Autumn 01

10

IA-64 Register Stack



- Eliminate the need for save / restore by reserving fixed blocks in register
- However, fixed blocks waste resources

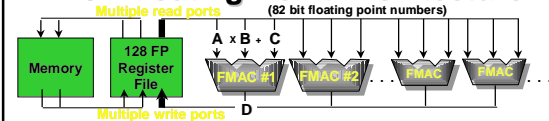
- IA-64 able to reserve variable block sizes
- No wasted resources

11/2/01

VLIW CSE 471 Autumn 01

11

IA-64 Floating-Point Architecture



- 128 registers
 - Allows parallel execution of multiple floating-point operations
- Simultaneous Multiply - Accumulate (FMAC)
 - 3-input, 1-output operation : $a * b + c = d$
 - Shorter latency than independent multiply and add
 - Greater internal precision and single rounding error

11/2/01

VLIW CSE 471 Autumn 01

12

Predication Basic Idea

- Associate a Boolean condition (predicate) with the issue, execution, or commit of an instruction
 - The stage in which to test the predicate is an implementation choice
- If the predicate is true, the result of the instruction is kept
- If the predicate is false, the instruction is nullified
- Distinction between
 - **Partial predication**: only a few opcodes can be predicated
 - **Full predication**: every instruction is predicated

11/2/01

VLIW CSE 471 Autumn 01

13

Predication Benefits

- Allows compiler to overlap the execution of independent control constructs w/o code explosion
- Allows compiler to reduce frequency of branch instructions and, consequently, of branch mispredictions
- Reduces the number of branches to be tested in a given cycle
- Reduces the number of multiple execution paths and associated hardware costs (copies of register maps etc.)
- Allows code movement in superblocks

11/2/01

VLIW CSE 471 Autumn 01

14

Predication Costs

- Increased fetch utilization
- Increased register consumption
- If predication is tested at commit time, increased functional-unit utilization
- With code movement, increased complexity of exception handling
 - For example, insert extra instructions for exception checking

11/2/01

VLIW CSE 471 Autumn 01

15

Flavors of Predication Implementation

- Has its roots in vector machines like CRAY-1
 - Creation of vector masks to control vector operations on an element per element basis
- Often (partial) predication limited to **conditional moves** as, e.g., in the Alpha, MIPS 10000, Power PC, SPARC and the Pentium Pro
- **Full predication**: Every instruction predicated as in IA-64

11/2/01

VLIW CSE 471 Autumn 01

16

Partial Predication: Conditional Moves

- **CMOV R1, R2, R3**
 - Move R2 to R1 if R3 \neq 0
- Main compiler use: If (cond) S1 (with result in Rres)
 - (1) Compute result of S1 in Rs1;
 - (2) Compute condition in Rcond;
 - (3) CMOV Rres, Rs1, Rcond
- Increases register pressure (Rcond is general register)
- No need (in this example) for branch predication
- Very useful if condition can be computed ahead or, e.g., in parallel with result.

11/2/01

VLIW CSE 471 Autumn 01

17

Other Forms of Partial Predication

- **Select dest, src1, src2, cond**
 - Corresponds to C-like $dest = (cond) ? src1 : src2$
 - Note the destination register is always assigned a value
 - Use in the Multiflow (first commercial VLIW machine)
- **Nullify**
 - Any register-register instruction can nullify the next instruction, thus making it conditional

11/2/01

VLIW CSE 471 Autumn 01

18

Full Predication

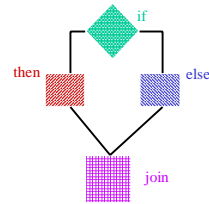
- Define predicates with instructions of the form:
 $\text{Pred}_{\langle \text{cmp} \rangle} \text{Pout1}_{\langle \text{type} \rangle}, \text{Pout2}_{\langle \text{type} \rangle}, \text{src1}, \text{src2} (\text{P}_{in})$ where
 - Pout1 and Pout2 are assigned values according to the comparison between src1 and src2 and the cmp “opcode”
 - The predicate types are most often U (unconditional) and \bar{U} its complement, and OR and \bar{OR}
 - The predicate define instruction can itself be predicated with the value of P_{in}
 - There are definite rules for that, e.g., if $\text{P}_{in} = 0$, U and \bar{U} are set to 0 independently of the result of the comparison and the OR predicates are not modified.

11/2/01

VLIW CSE 471 Autumn 01

19

If-conversion



The if condition will set p1 to U

The then will be executed predicated on p1(U)

The else will be executed predicated on p1(\bar{U})

The “join” will in general be predicated on some form of OR predicate

11/2/01

VLIW CSE 471 Autumn 01

20