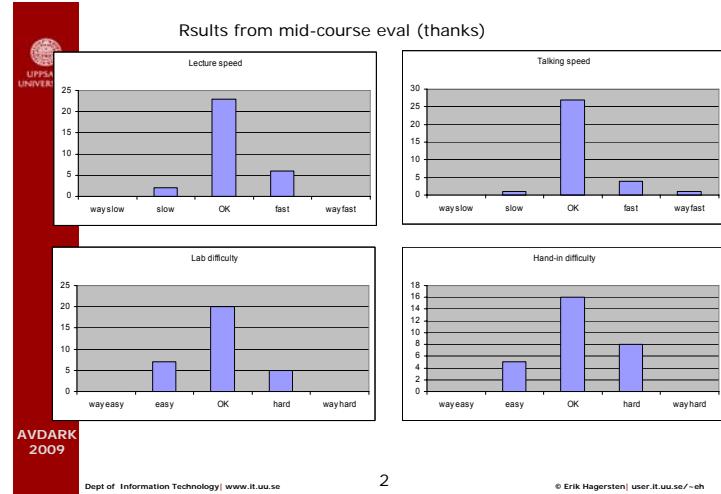
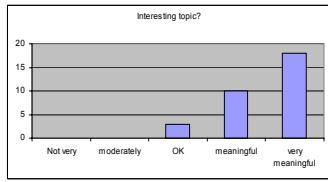


# CPU design options

Erik Hagersten  
Uppsala University



2



ALL NICE COMMENTS OMITTED (BUT THANKS!!)  
Repetition of prev lect is good, but slides should then need more details  
Great course, but needs more background/prerequisite  
More real experience stuff!  
Slides fg color does not work for B/W. Bringing real HW is fun!  
Need guidance for exam and what to study  
Hard to follow the theory  
Slides need more annotations (x2)  
More on current research, such as what is the current MP systems (x2)  
There are schedule conflicts w other courses (+ slides need more...) x2  
I miss Mr Frog (x4)  
Good labs, maybe need somewhat better descriptions  
The lecture room is too cold  
Lower standard to pass hand-in would be better  
Overlap with other course. Lab hand-in vary good  
I lack a compendium. Labs and assignment are important for comprehension.

AVDARK  
2009

Dept of Information Technology | www.it.uu.se

© Erik Hagersten | user.it.uu.se/~eh



## Schedule in a nutshell

1. **Memory Systems** (~Appendix C in 4th Ed)  
Caches, VM, DRAM, microbenchmarks, optimizing SW
2. **Multiprocessors**  
TLP: coherence, memory models, synchronization
3. **Scalable Multiprocessors**  
Scalability, implementations, programming, ...
4. **CPUs**  
ILP: pipelines, scheduling, superscalars, VLIWs, SIMD instructions...
5. **Widening + Future** (~Chapter 1 in 4th Ed)  
Technology impact, GPUs, Network processors, **Multicores (!!)**

AVDARK  
2009

Dept of Information Technology | www.it.uu.se

4

© Erik Hagersten | user.it.uu.se/~eh



## Goal for this course

- Understand **how and why** modern computer systems are designed the way they are:
  - ⇒ • pipelines
  - ✓ memory organization
  - ✓ virtual/physical memory ...
- Understand **how and why** multiprocessors are built
  - ✓ Cache coherence
  - ✓ Memory models
  - ✓ Synchronization...
- Understand **how and why** parallelism is created and
  - ⇒ • Instruction-level parallelism
  - ⇒ • Memory-level parallelism
  - ✓ Thread-level parallelism...
- Understand **how and why** multiprocessors of combined SIMD/MIMD type are built
  - GPU
  - ⇒ • Vector processing...
- Understand **how** computer systems are adopted to different usage areas
  - General-purpose processors
  - Embedded/network processors...
- Understand the physical limitation of modern computers
  - Bandwidth
  - Energy
  - Cooling...

AVDARK  
2009

Dept of Information Technology | www.it.uu.se

5

© Erik Hagersten | user.it.uu.se/~eh



## How it all started...the fossils

- ENIAC J.P. Eckert and J. Mauchly, Univ. of Pennsylvania, WW2
  - Electro Numeric Integrator And Calculator, 18.000 vacuum tubes
- EDVAC, J. V Neumann, operational 1952
  - Electric Discrete Variable Automatic Computer (stored programs)
- EDSAC, M. Wilkes, Cambridge University, 1949
  - Electric Delay Storage Automatic Calculator
- Mark-I... H. Aiken, Harvard, WW2, Electro-mechanic
- K. Zuse, Germany, electromech. computer, special purpose, WW2
- BARK, KTH, Gösta Neovius (was et Ericsson), Electro-mechanic early 50s
- BESK, KTH, Erik Stemme (was at Chalmers) early 50s
- SMIL, LTH mid 50s

AVDARK  
2009

Dept of Information Technology | www.it.uu.se

6

© Erik Hagersten | user.it.uu.se/~eh

## How do you tell a good idea from a bad

The Book: The performance-centric approach

- CPI = #execution-cycles / #instructions executed (~ISA goodness – lower is better)
- CPI \* cycle time → performance
- CPI = CPI<sub>CPU</sub> + CPI<sub>Mem</sub>

*The book rarely covers other design tradeoffs*

- The feature centric approach...
- The cost-centric approach...
- Energy-centric approach...
- Verification-centric approach...

7

© Erik Hagersten| user.it.uu.se/~eh

## The Book: Quantitative methodology

Make design decisions based on execution statistics.

Select workloads (programs representative for usage)

Instruction mix measurements: statistics of relative usage of different components in an ISA

Experimental methodologies

- Profiling through tracing
- ISA simulators

8

© Erik Hagersten| user.it.uu.se/~eh

## Two guiding stars -- the RISC approach:

Make the common case fast

- Simulate and profile anticipated execution
- Make cost-functions for features
- Optimize for overall end result (end performance)

Watch out for Amdahl's law

- Speedup = Execution\_time<sub>OLD</sub> / Execution\_time<sub>NEW</sub>
- [(1-Fraction<sub>ENHANCED</sub>) + Fraction<sub>ENHANCED</sub> / Speedup<sub>ENHANCED</sub>]

9

© Erik Hagersten| user.it.uu.se/~eh

## Instruction Set Architecture (ISA)

-- the interface between software and hardware.

Tradeoffs between many options:

- functionality for OS and compiler
  - wish for many addressing modes
  - compact instruction representation
  - format compatible with the memory system of choice
  - desire to last for many generations
  - bridging the semantic gap (old desire...)
- RISC: the biggest “customer” is the compiler

10

© Erik Hagersten| user.it.uu.se/~eh

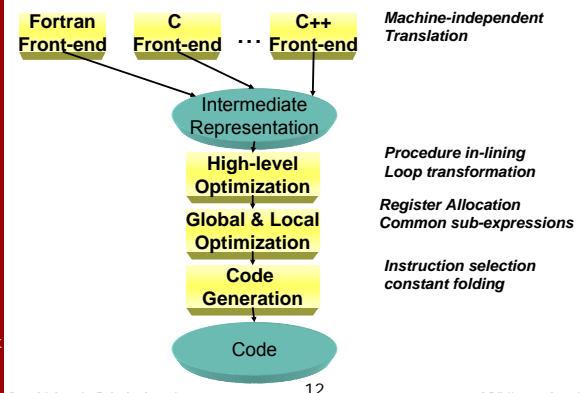
## ISA trends today

- CPU families built around “Instruction Set Architectures” ISA
- Many incarnations of the same ISA
- ISAs lasting longer (~10 years)
- Consolidation in the market - fewer ISAs (not for embedded...)
- 15 years ago ISAs were driven by academia
- Today ISAs technically do not matter all that much (market-driven)
- How many of you will ever design an ISA?
- How many ISAs will be designed in Sweden?

11

© Erik Hagersten| user.it.uu.se/~eh

## Compiler Organization

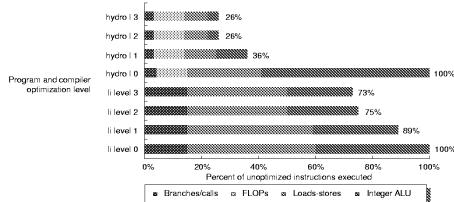


12

© Erik Hagersten| user.it.uu.se/~eh

## Compilers – a moving target!

### The impact of compiler optimizations



- Compiler optimizations affect the number of instructions as well as the distribution of executed instructions (the instruction mix)

AVDARK 2009

Dept of Information Technology | www.it.uu.se

13

© Erik Hagersten | user.it.uu.se/~eh

## Memory allocation model also has a huge impact

### Stack

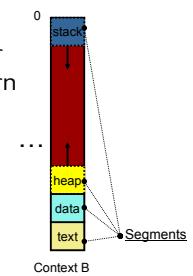
- local variables in activation record
- addressing relative to stack pointer
- stack pointer modified on call/return

### Global data area

- large constants
- global static structures

### Heap

- dynamic objects
- often accessed through pointers



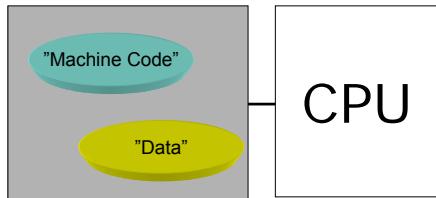
AVDARK 2009

Dept of Information Technology | www.it.uu.se

14

© Erik Hagersten | user.it.uu.se/~eh

## Execution in a CPU



AVDARK 2009

Dept of Information Technology | www.it.uu.se

15

© Erik Hagersten | user.it.uu.se/~eh

## Operand models

Example: C := A + B

Stack      Accumulator      Register

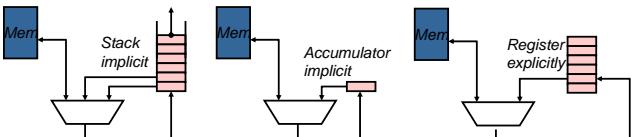
|          |           |              |
|----------|-----------|--------------|
| PUSH [A] | LOAD [A]  | LOAD R1,[A]  |
| PUSH [B] | ADD [B]   | ADD R1,[B]   |
| ADD      | STORE [C] | STORE [C],R1 |
| POP [C]  |           |              |

AVDARK 2009

Dept of Information Technology | www.it.uu.se

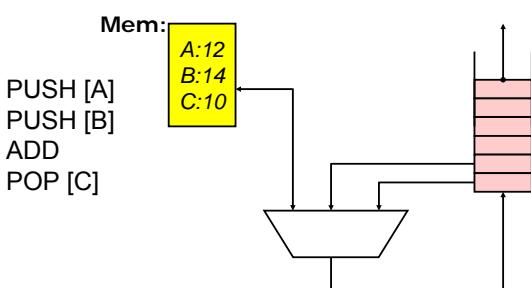
16

© Erik Hagersten | user.it.uu.se/~eh



## Stack-based machine

Example: C := A + B



AVDARK 2009

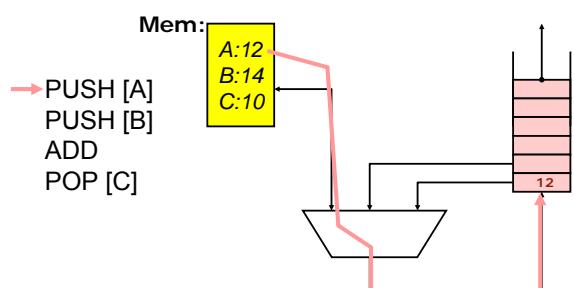
Dept of Information Technology | www.it.uu.se

17

© Erik Hagersten | user.it.uu.se/~eh

## Stack-based machine

Example: C := A + B



AVDARK 2009

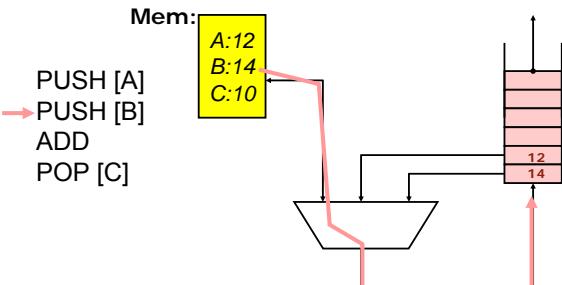
Dept of Information Technology | www.it.uu.se

18

© Erik Hagersten | user.it.uu.se/~eh

## Stack-based machine

Example:  $C := A + B$

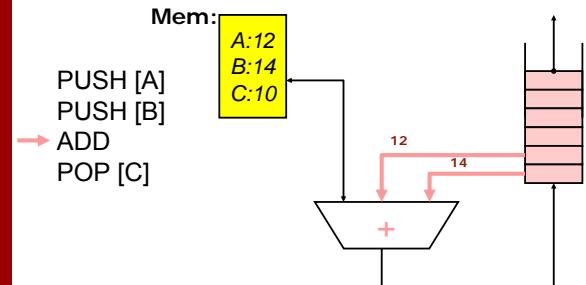


19

© Erik Hagersten | user.it.uu.se/~eh

## Stack-based machine

Example:  $C := A + B$

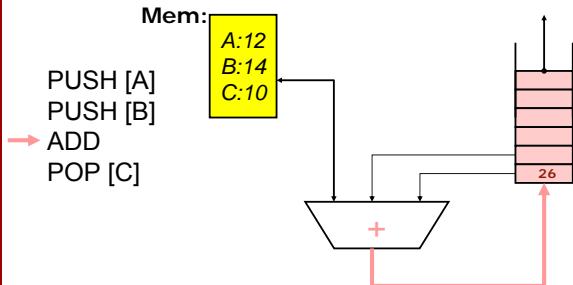


20

© Erik Hagersten | user.it.uu.se/~eh

## Stack-based machine

Example:  $C := A + B$

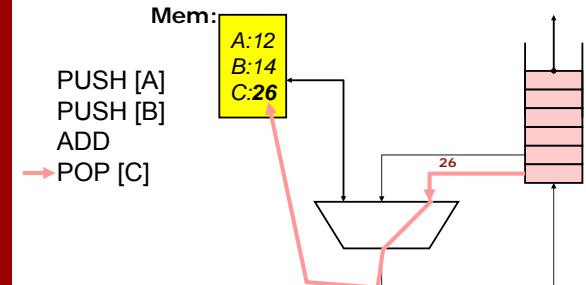


21

© Erik Hagersten | user.it.uu.se/~eh

## Stack-based machine

Example:  $C := A + B$



22

© Erik Hagersten | user.it.uu.se/~eh

## Stack-based

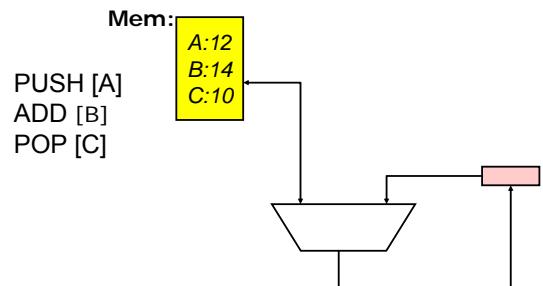
- Implicit operands
- Compact code format (1 instr. = 1byte)
- Simple to implement
- Not optimal for speed!!!

23

© Erik Hagersten | user.it.uu.se/~eh

## Accumulator-based

≈ Stack-based with a depth of one  
One implicit operand from the accumulator

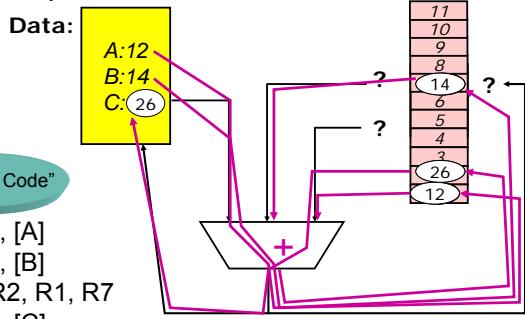


24

© Erik Hagersten | user.it.uu.se/~eh

## Register-based machine

Example: C := A + B



AVDARK  
2009

Dept of Information Technology | www.it.uu.se

25

© Erik Hagersten | user.it.uu.se/~eh

## Register-based

- Commercial success:
  - CISC: X86
  - RISC: (Alpha), SPARC, (HP-PA), Power, MIPS, ARM
  - VLIW: IA64
- Explicit operands (i.e., "registers")
- Wasteful instr. format (1instr. = 4bytes)
- Suits optimizing compilers
- Optimal for speed!!!

AVDARK  
2009

Dept of Information Technology | www.it.uu.se

26

© Erik Hagersten | user.it.uu.se/~eh

## Properties of operand models

|             | Compiler Construction | Implementation Efficiency | Code Size |
|-------------|-----------------------|---------------------------|-----------|
| Stack       | +                     | --                        | ++        |
| Accumulator | --                    | -                         | +         |
| Register    | ++                    | ++                        | --        |

AVDARK  
2009

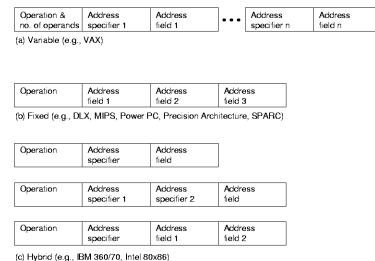
General-purpose register model dominates today

Reason: general model for compilers and efficient implementation

27

© Erik Hagersten | user.it.uu.se/~eh

## Instruction formats



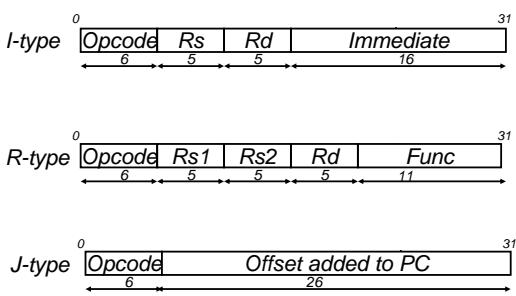
AVDARK  
2009

Dept of Information Technology | www.it.uu.se

28

© Erik Hagersten | user.it.uu.se/~eh

## Generic Instruction Formats



AVDARK  
2009

Dept of Information Technology | www.it.uu.se

29

© Erik Hagersten | user.it.uu.se/~eh

## Generic instructions (Load/Store Architecture)

| Instruction type | Example       | Meaning                            |
|------------------|---------------|------------------------------------|
| Load             | LW R1,30(R2)  | Regs[R1] ← Mem[30+Regs[R2]]        |
| Store            | SW 30(R2),R1  | Mem[30+Regs[R2]] ← Regs[R1]        |
| ALU              | ADD R1,R2,R3  | Regs[R1] ← Regs[R2] + Regs[R3]     |
| Control          | BEQZ R1,KALLE | if (Regs[R1]==0)<br>PC ← KALLE + 4 |

AVDARK  
2009

Dept of Information Technology | www.it.uu.se

30

© Erik Hagersten | user.it.uu.se/~eh

## Generic ALU Instructions

- Integer arithmetic
  - [add, sub] x [signed, unsigned] x [register, immediate]
  - e.g., ADD, ADDI, ADDU, ADDUI, SUB, SUBI, SUBU, SUBUI
- Logical
  - [and, or, xor] x [register, immediate]
  - e.g., AND, ANDI, OR, ORI, XOR, XRI
- Load upper half immediate load
  - It takes two instructions to load a 32 bit immediate

Dept of Information Technology | www.it.uu.se

31

© Erik Hagersten | user.it.uu.se/~eh

## Generic FP Instructions

- Floating Point arithmetic
  - [add, sub, mult, div] x [double, single]
  - e.g., ADDD, ADDF, SUBD, SUBF, ...
- Compares (sets "compare bit")
  - [lt, gt, le, ge, eq, ne] x [double, immediate]
  - e.g., LTD, GEF, ...
- Convert from/to integer, Fpregs
  - CVTF2I, CVTF2D, CVTI2D, ...

Dept of Information Technology | www.it.uu.se

32

© Erik Hagersten | user.it.uu.se/~eh

## Simple Control

- Branches if equal or if not equal
  - BEQZ, BNEZ, cmp to register,  
PC := PC + 4 + immediate<sub>16</sub>
  - BFPT, BFPF, cmp to "FP compare bit",  
PC := PC + 4 + immediate<sub>16</sub>
- Jumps
  - J: Jump --  
PC := PC + immediate<sub>26</sub>
  - JAL: Jump And Link --  
R31 := PC + 4; PC := PC + immediate<sub>26</sub>
  - JALR: Jump And Link Register --  
R31 := PC + 4; PC := PC + Reg
  - JR: Jump Register –  
PC := PC + Reg ("return from JAL or JALR")

Dept of Information Technology | www.it.uu.se

33

© Erik Hagersten | user.it.uu.se/~eh

## Conditional Branches

### Three options:

- Condition Code: Most operations have "side effects" on set of CC-bits. A branch depends on some CC-bit
- Condition Register. A named register is used to hold the result from a compare instruction. A following branch instruction names the same register.
- Compare and Branch. The compare and the branch is performed in the same instruction.

Dept of Information Technology | www.it.uu.se

34

© Erik Hagersten | user.it.uu.se/~eh

## Important Operand Modes

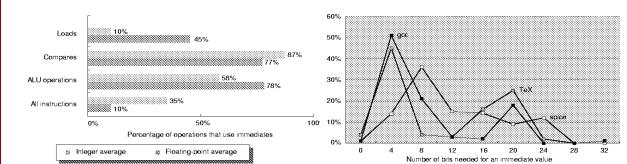
| Addressing mode | Example instruction | Meaning                                   | When used                  |
|-----------------|---------------------|---|----------------------------|
| Immediate       | Add R3, R4,#3       | Regs[R3] ← Regs[R4]+ 3                    | For constants.             |
| Displacement    | Add R3, R4,100(R1)  | Regs[R3] ← Regs[R4]+<br>Mem[100+Regs[R1]] | Accessing local variables. |

Dept of Information Technology | www.it.uu.se

35

© Erik Hagersten | user.it.uu.se/~eh

## Size of immediates



- ♦ Immediate operands are very important for ALU and compare operations
- ♦ 16-bit immediates seem sufficient (75%-80%)

Dept of Information Technology | www.it.uu.se

36

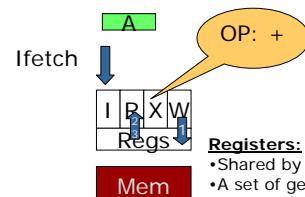
© Erik Hagersten | user.it.uu.se/~eh

# Implementing ISAs --pipelines

Erik Hagersten  
Uppsala University

## EXAMPLE: pipeline implementation

### Add R1, R2, R3



#### Registers:

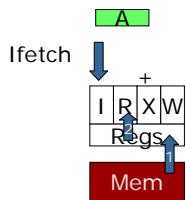
- Shared by all pipeline stages
- A set of general purpose registers (GPRs)
- Some specialized registers (e.g., PC)

38

© Erik Hagersten| user.it.uu.se/~eh

### Load Operation:

**LD R1, mem[cnst+R2]**



AVDARK  
2009

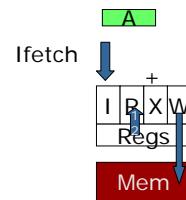
Dept of Information Technology| www.it.uu.se

39

© Erik Hagersten| user.it.uu.se/~eh

### Store Operation:

**ST mem[cnst+R1], R2**



AVDARK  
2009

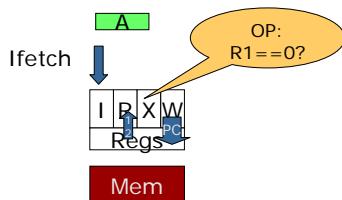
Dept of Information Technology| www.it.uu.se

40

© Erik Hagersten| user.it.uu.se/~eh

### EXAMPLE: Branch to R2 if R1 == 0

**BEQZ R1, R2**



AVDARK  
2009

Dept of Information Technology| www.it.uu.se

41

© Erik Hagersten| user.it.uu.se/~eh

### Initially

|   |                       |
|---|-----------------------|
| D | IF RegC < 100 GOTO A  |
| C | RegC := RegC + 1      |
| B | RegB := RegA + 1      |
| A | LD RegA, (100 + RegC) |

PC →



AVDARK  
2009

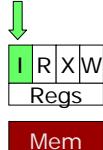
Dept of Information Technology| www.it.uu.se

42

© Erik Hagersten| user.it.uu.se/~eh

## Cycle 1

**D** IF RegC < 100 GOTO A  
**C** RegC := RegC + 1  
**B** RegB := RegA + 1  
**A** LD RegA, (100 + RegC)



AVDARK  
2009

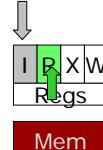
Dept of Information Technology | www.it.uu.se

43

© Erik Hagersten | user.it.uu.se/~eh

## Cycle 2

**D** IF RegC < 100 GOTO A  
**C** RegC := RegC + 1  
**B** RegB := RegA + 1  
**A** LD RegA, (100 + RegC)



AVDARK  
2009

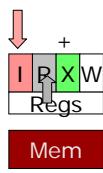
Dept of Information Technology | www.it.uu.se

44

© Erik Hagersten | user.it.uu.se/~eh

## Cycle 3

**D** IF RegC < 100 GOTO A  
**C** RegC := RegC + 1  
**B** RegB := RegA + 1  
**A** LD RegA, (100 + RegC)



AVDARK  
2009

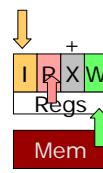
Dept of Information Technology | www.it.uu.se

45

© Erik Hagersten | user.it.uu.se/~eh

## Cycle 4

**D** IF RegC < 100 GOTO A  
**C** RegC := RegC + 1  
**B** RegB := RegA + 1  
**A** LD RegA, (100 + RegC)



AVDARK  
2009

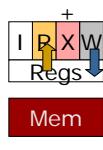
Dept of Information Technology | www.it.uu.se

46

© Erik Hagersten | user.it.uu.se/~eh

## Cycle 5

**D** IF RegC < 100 GOTO A  
**C** RegC := RegC + 1  
**B** RegB := RegA + 1  
**A** LD RegA, (100 + RegC)



AVDARK  
2009

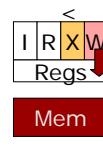
Dept of Information Technology | www.it.uu.se

47

© Erik Hagersten | user.it.uu.se/~eh

## Cycle 6

**D** IF RegC < 100 GOTO A  
**C** RegC := RegC + 1  
**B** RegB := RegA + 1  
**A** LD RegA, (100 + RegC)



AVDARK  
2009

Dept of Information Technology | www.it.uu.se

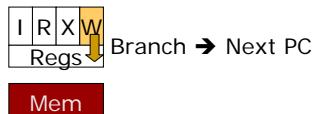
48

© Erik Hagersten | user.it.uu.se/~eh

## Cycle 7

PC →

|      |                       |
|------|-----------------------|
| FFFF | IF RegC < 100 GOTO A  |
| FFFF | RegC := RegC + 1      |
| FFFF | RegB := RegA + 1      |
| FFFF | LD RegA, (100 + RegC) |



AVDARK  
2009

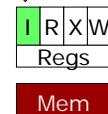
Dept of Information Technology | www.it.uu.se

49

© Erik Hagersten | user.it.uu.se/~eh

## Cycle 8

|   |                       |
|---|-----------------------|
| D | IF RegC < 100 GOTO A  |
| C | RegC := RegC + 1      |
| B | RegB := RegA + 1      |
| A | LD RegA, (100 + RegC) |



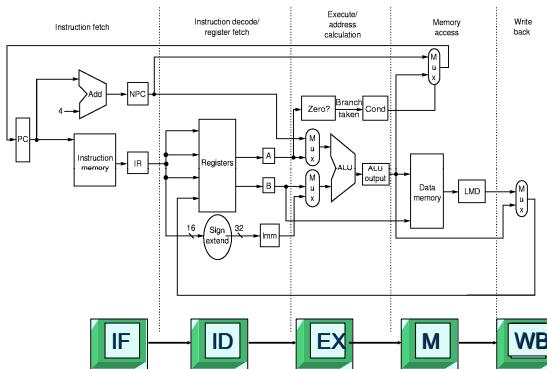
AVDARK  
2009

Dept of Information Technology | www.it.uu.se

50

© Erik Hagersten | user.it.uu.se/~eh

## Example: 5-stage pipeline



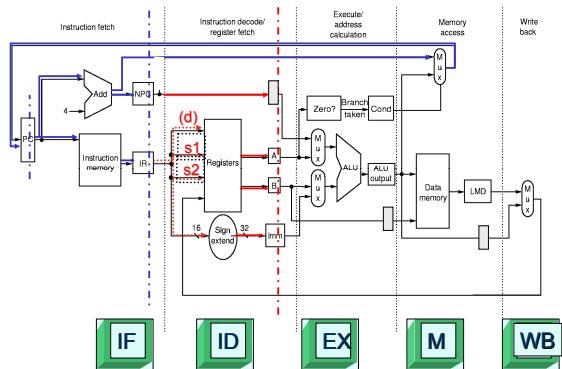
AVDARK  
2009

Dept of Information Technology | www.it.uu.se

51

© Erik Hagersten | user.it.uu.se/~eh

## Example: 5-stage pipeline



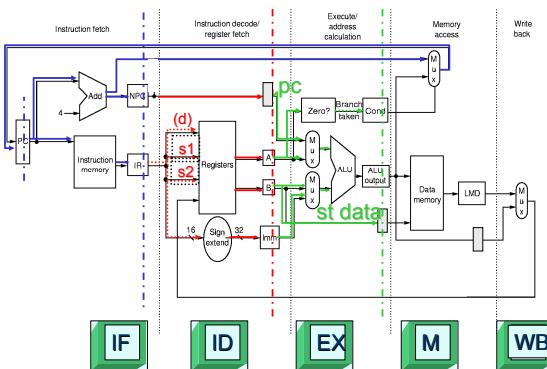
AVDARK  
2009

Dept of Information Technology | www.it.uu.se

52

© Erik Hagersten | user.it.uu.se/~eh

## Example: 5-stage pipeline



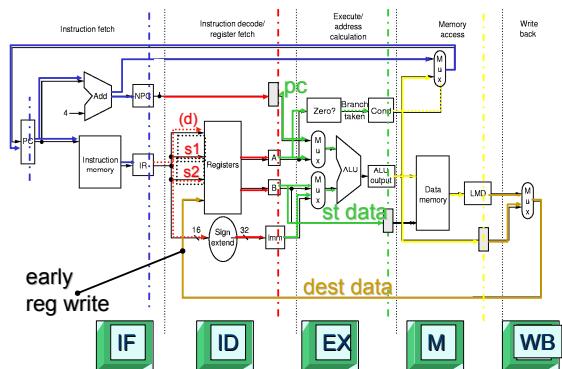
AVDARK  
2009

Dept of Information Technology | www.it.uu.se

53

© Erik Hagersten | user.it.uu.se/~eh

## Example: 5-stage pipeline



AVDARK  
2009

Dept of Information Technology | www.it.uu.se

54

© Erik Hagersten | user.it.uu.se/~eh

## Fundamental limitations

Hazards prevent instructions from executing in parallel:

**Structural hazards:** Simultaneous use of same resource  
If unified I+D\$: LW will conflict with later I-fetch

**Data hazards:** Data dependencies between instructions  
LW R1, 100(R2) /\* result avail in 2 - 100 cycles \*/  
ADD R5, R1, R7

**Control hazards:** Change in program flow  
BNEQ R1, #OFFSET  
ADD R5, R2, R3

**Serialization of the execution by stalling the pipeline is one, although inefficient, way to avoid hazards**

## Fundamental types of data hazards

Code sequence:  $Op_i A$   
 $Op_{i+1} A$

**RAW (Read-After-Write)**

$Op_{i+1}$  reads A before  $Op_i$  modifies A.  $Op_{i+1}$  reads old A!

**WAR (Write-After-Read)**

$Op_{i+1}$  modifies A before  $Op_i$  reads A.  
 $Op_i$  reads new A

**WAW (Write-After-Write)**

$Op_{i+1}$  modifies A before  $Op_i$ .  
The value in A is the one written by  $Op_i$ , i.e., an old A.

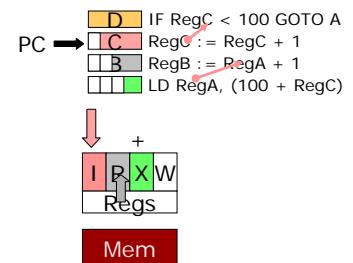
## Hazard avoidance techniques

**Static techniques (compiler):** code scheduling to avoid hazards

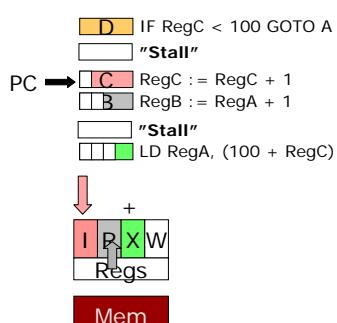
**Dynamic techniques:** hardware mechanisms to eliminate or reduce impact of hazards (e.g., out-of-order stuff)

**Hybrid techniques:** rely on compiler as well as hardware techniques to resolve hazards (e.g. VLIW support – later)

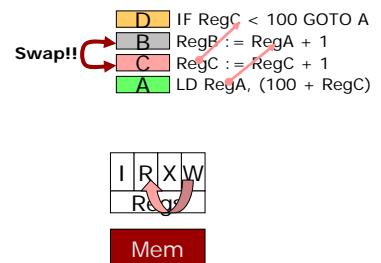
## Cycle 3



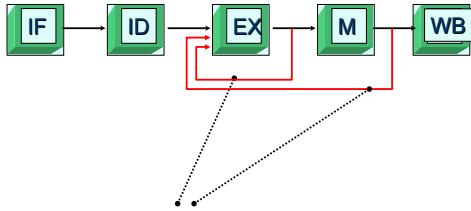
## Cycle 3



## Fix alt1: code scheduling



## Fix alt2: Bypass hardware



- **Forwarding (or bypassing):** provides a direct path from M and WB to EX
- Only helps for ALU ops. What about load operations?

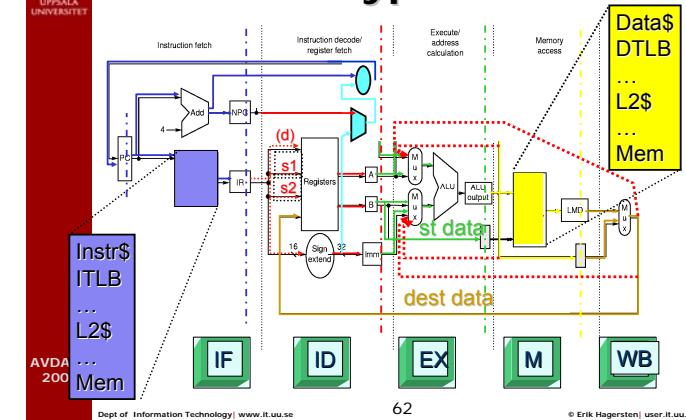
AVDARK 2009

Dept of Information Technology | www.it.uu.se

61

© Erik Hagersten | user.it.uu.se/~eh

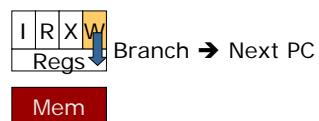
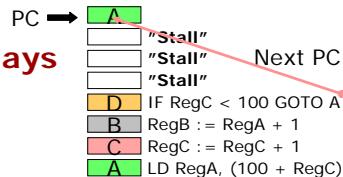
## DLX with bypass



62

© Erik Hagersten | user.it.uu.se/~eh

## Branch delays



8 cycles per iteration of 4 instructions ☺  
Need longer basic blocks with independent instr.

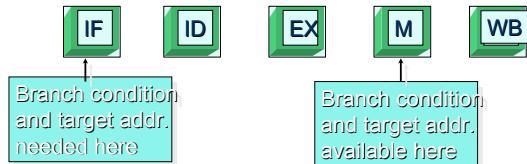
AVDARK 2009

Dept of Information Technology | www.it.uu.se

63

© Erik Hagersten | user.it.uu.se/~eh

## Avoiding control hazards



Duplicate resources in ALU to compute branch condition and branch target address earlier

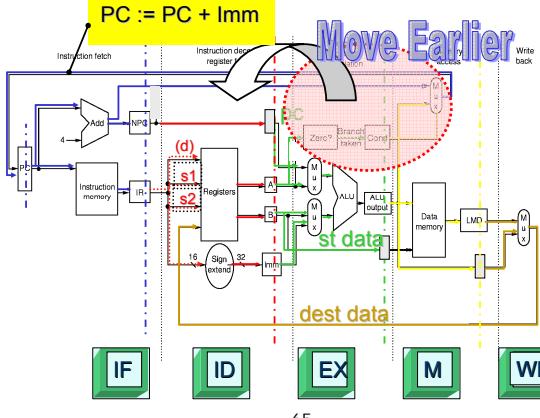
Branch delay cannot be completely eliminated

Branch prediction and code scheduling can reduce the branch penalty

64

© Erik Hagersten | user.it.uu.se/~eh

## Fix1: Minimizing Branch Delay Effects



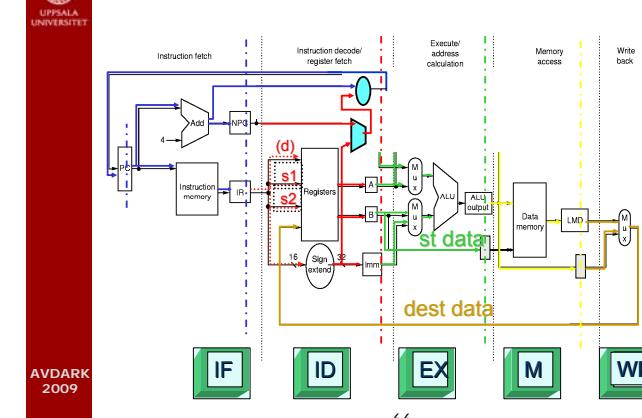
AVDARK 2009

Dept of Information Technology | www.it.uu.se

65

© Erik Hagersten | user.it.uu.se/~eh

## Fix1: Minimizing Branch Delay Effects



66

© Erik Hagersten | user.it.uu.se/~eh

## Fix2: Static tricks

Delayed branch (schedule useful instr. in delay slot)

- Define branch to take place after a following instruction
- CONS: this is visible to SW, i.e., forces compatibility between generations

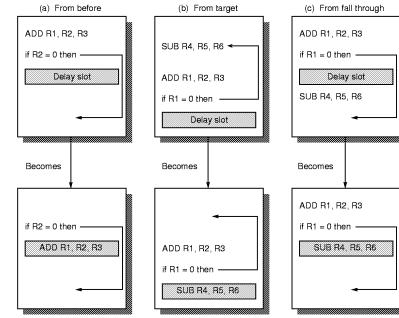
Predict Branch not taken (a fairly rare case)

- Execute successor instructions in sequence
- "Squash" instructions in pipeline if the branch is actually taken
- Works well if state is updated late in the pipeline
- 30%-38% of conditional branches are not taken on average

Predict Branch taken (a fairly common case)

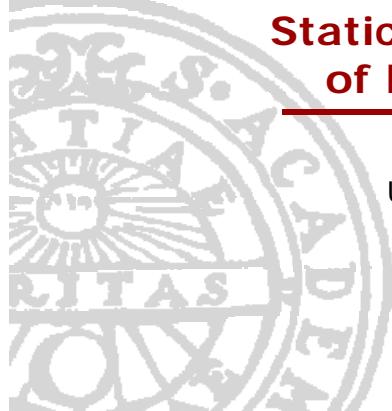
- 62%-70% of conditional branches are taken on average
- Does not make sense for the generic arch. but may do for other pipeline organizations

## Static scheduling to avoid stalls



Dynamic solution  
[Goto 89](#)

- Scheduling an instruction from before is always safe
- Scheduling from target or from the not-taken path is not always safe; must be guaranteed that speculative instr. do no harm.



## Static Scheduling of Instructions

Erik Hagersten  
Uppsala University  
Sweden

## Scheduling example

```
for (i=1; i<=1000; i=i+1)
    x[i] = x[i] + 10;

Iterations are independent => parallel execution

loop:   LD      F0, O(R1)      ; F0 = array element
        ADDD   F4, F0, F2      ; Add scalar constant
        SD      O(R1), F4       ; Save result
        SUBI   R1, R1, #8       ; decrement array ptr.
        BNEZ   R1, loop         ; reiterate if R1 != 0
```

Can we eliminate all penalties in each iteration?  
How about moving SD down?

## Scheduling in each loop iteration

|               |  |
|---------------|--|
| Original loop |  |
| loop:         | LD      F0, O(R1)<br><i>stall</i><br>ADDD  F4, F0, F2<br><i>stall</i><br><i>stall</i><br>SD      O(R1), F4<br>SUBI  R1, R1, #8<br>BNEZ  R1, loop<br><i>stall</i> |

5 instructions + 4 bubbles = 9 cycles / iteration  
(~one cycle per iteration on a vector architecture)

Can we do better by scheduling across iterations?

## Scheduling in each loop iteration

| Original loop   | Statically scheduled loop |
|-----------------|---------------------------|
| <b>loop:</b>    | <b>loop:</b>              |
| LD F0, O(R1)    | LD F0, O(R1)              |
| stall           | stall                     |
| ADDD F4, F0, F2 | ADDD F4, F0, F2           |
| stall           | stall                     |
| stall           | BNEZ R1, loop             |
| SD O(R1), F4    | SD 8(R1), F4              |
| SUBI R1, R1, #8 |                           |
| BNEZ R1, loop   |                           |
| stall           |                           |

5 instruction + 4 bubbles = 9c / iteration      5 instruction + 1 bubble = 6c / iteration

AVDARK  
2009

Dept of Information Technology | www.it.uu.se

73

© Erik Hagersten | user.it.uu.se/~eh

Can we do even better by scheduling across iterations?

## Unoptimized loop unrolling 4x

| loop: | LD F0, O(R1)      | ; drop SUBI & BNEZ |
|-------|-------------------|--------------------|
|       | stall             |                    |
|       | ADDD F4, F0, F2   |                    |
|       | stall             |                    |
|       | SD O(R1), F4      |                    |
|       | LD F6, -8(R1)     |                    |
|       | stall             |                    |
|       | ADDD F8, F6, F2   |                    |
|       | stall             |                    |
|       | SD -8(R1), F8     |                    |
|       | LD F10, -16(R1)   |                    |
|       | stall             |                    |
|       | ADDD F12, F10, F2 |                    |
|       | stall             |                    |
|       | SD -16(R1), F12   |                    |
|       | LD F14, -24(R1)   |                    |
|       | stall             |                    |
|       | ADDD F16, F14, F2 |                    |
|       | SUBI R1, R1, #32  |                    |
|       | BNEZ R1, loop     |                    |
|       | SD -24(R1), F16   |                    |

24c / 4 iterations = 6 c / iteration

AVDARK  
2009

Dept of Information Technology | www.it.uu.se

74

© Erik Hagersten | user.it.uu.se/~eh

## Optimized scheduled unrolled loop

| <u>Important steps:</u> |  |
|-------------------------|--|
| loop:                   | LD F0, O(R1) Push loads up   |
|                         | LD F6, -8(R1) Push stores down   |
|                         | LD F10, -16(R1) Note: the displacement of the last store must be changed |
|                         | LD F14, -24(R1)  |
|                         | ADDD F4, F0, F2  |
|                         | ADDD F8, F6, F2  |
|                         | ADDD F12, F10, F2  |
|                         | ADDD F16, F14, F2  |
|                         | SD O(R1), F4   |
|                         | SD -8(R1), F8  |
|                         | SD -16(R1), F12  |
|                         | SUBI R1, R1, #32   |
|                         | BNEZ R1, loop  |
|                         | SD 8(R1), F16  |

All penalties are eliminated. CPI=1  
 14 cycles / 4 iterations => 3.5 cycles / iteration  
 From 9c to 3.5c per iteration => speedup 2.6

AVDARK  
2009

Dept of Information Technology | www.it.uu.se

© Erik Hagersten | user.it.uu.se/~eh

75

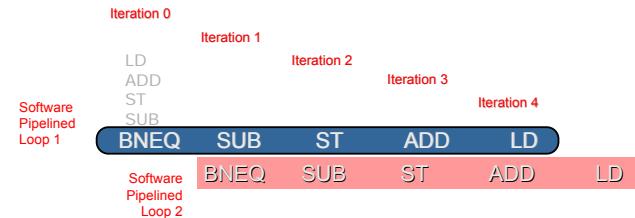
© Erik Hagersten | user.it.uu.se/~eh

© Erik Hagersten | user.it.uu.se/~eh

## Software pipelining 1(3)

### Symbolic loop unrolling

- The instructions in a loop are taken from different iterations in the original loop



AVDARK  
2009

Dept of Information Technology | www.it.uu.se

76

© Erik Hagersten | user.it.uu.se/~eh

## Software pipelining 2(3)

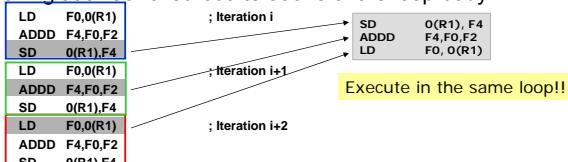
Example:

loop:

|               |  |
|---------------|--|
| LD F0,O(R1)   |  |
| ADDD F4,F0,F2 |  |
| SD O(R1),F4   |  |
| SUBI R1,R1,#8 |  |
| BNEZ R1,loop  |  |

AVDARK  
2009

Looking at three rolled-out iterations of the loop body:



77

© Erik Hagersten | user.it.uu.se/~eh

AVDARK  
2009

Dept of Information Technology | www.it.uu.se

78

© Erik Hagersten | user.it.uu.se/~eh

## Software pipelining 3(3)

Instructions from three consecutive iterations form the loop body:

< prologue code >

|       |                                    |
|-------|------------------------------------|
| loop: | SD O(R1),F4 ; from iteration i     |
|       | ADDD F4,F0,F2 ; from iteration i+1 |
|       | LD F0,-16(R1) ; from iteration i+2 |
|       | SUBI R1,R1,#8                      |
|       | BNEZ R1,loop                       |

< prologue code >

- No data dependencies *within* a loop iteration
- The dependence distance is 1 iterations
- WAR hazard elimination is needed (register renaming)
- 5c / iteration, but only uses 2 FP reg (instead of 8)

## Software pipelining

- "Symbolic Loop Unrolling"
- Very tricky for complicated loops
- Less code expansion than outlining
- Register-poor if "rotating" is used
- Needed to hide large latencies (see IA-64)

## Dependencies: Revisited

Two instructions must be **independent** in order to execute in parallel

- Three classes of dependencies that limit parallelism:

- Data dependencies

$X := ...$

$.... := ... X ....$

- Name dependencies

$... := ... X$

$X := ...$

- Control dependencies

If ( $X > 0$ ) then

$Y := ...$

## Getting desperate for ILP

Erik Hagersten  
Uppsala University  
Sweden

## Multiple instruction issue per clock

Goal: Extracting ILP so that CPI < 1 , i.e., IPC > 1

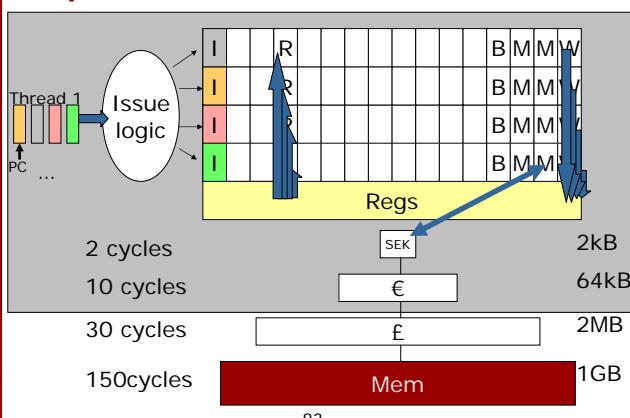
### Superscalar:

- Combine static and dynamic scheduling to issue multiple instructions per clock
- HW finds independent instructions in "sequential" code
- Predominant: (PowerPC, SPARC, Alpha, HP-PA)

### Very Long Instruction Words (VLIW):

- Static scheduling used to form packages of independent instructions that can be issued together
- Relies on compiler to find independent instructions (IA-64)

## Superscalars



## Example: A Superscalar DLX

- Issue 2 instructions simultaneously: 1 FP & 1 integer
  - Fetch 64-bits/clock cycle; Integer instr. on left, FP on right
  - Can only issue 2nd instruction if 1st instruction issues
  - Need more ports to the register file

| Type | Pipe stages |    |    |     |     |     |
|------|-------------|----|----|-----|-----|-----|
|      | IF          | ID | EX | MEM | WB  |     |
| Int. | IF          | ID | EX | MEM | WB  |     |
| FP   | IF          | ID | EX | MEM | WB  |     |
| Int. |             | IF | ID | EX  | MEM | WB  |
| FP   |             | IF | ID | EX  | MEM | WB  |
| Int. |             |    | IF | ID  | EX  | MEM |
| FP   |             |    | IF | ID  | EX  | MEM |

• EX stage should be fully pipelined

• 1 load delay slot corresponds to three instructions!

## Statically Scheduled Superscalar DLX

| Integer instruction | FP instruction    | Clock cycle |
|---------------------|-------------------|-------------|
| Loop:               |                   |             |
| LD F0, 0(R1)        |                   | 1           |
| LD F6, -8(R1)       |                   | 2           |
| LD F10, -16(R1)     | ADDD F4, F0, F2   | 3           |
| LD F14, -24(R1)     | ADDD F8, F6, F2   | 4           |
| LD F18, -32(R1)     | ADDD F12, F10, F2 | 5           |
| SD 0(R1), F4        | ADDD F16, F14, F2 | 6           |
| SD -8(R1), F8       | ADDD F20, F18, F2 | 7           |
| SD -16(R1), F12     |                   | 8           |
| SD -24(R1), F16     |                   | 9           |
| SUBI R1, R1, #40    |                   | 10          |
| BNEZ R1, LOOP       |                   | 11          |
| SD -32(R1), F20     |                   | 12          |

Can be scheduled dynamically with Tomasulo's alg.

Issue: Difficult to find a sufficient number of instr. to issue

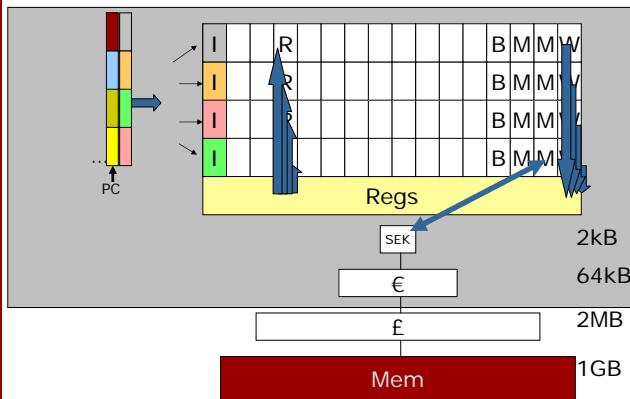
## Limits to superscalar execution

- Difficulties in scheduling within the constraints on number of functional units and the ILP in the code chunk
- Instruction decode complexity increases with the number of issued instructions
- Data and control dependencies are in general more costly in a superscalar processor than in a single-issue processor

Techniques to enlarge the instruction window to extract more ILP are important

Simple superscalars relying on compiler instead of HW complexity → VLIW

## VLIW: Very Long Instruction Word



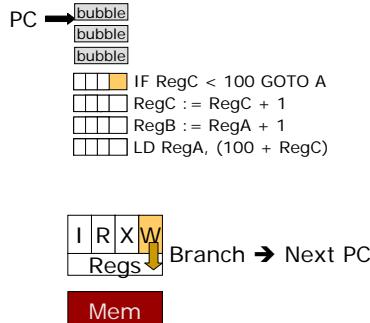
## Very Long Instruction Word (VLIW)

Compiler is responsible for instruction scheduling

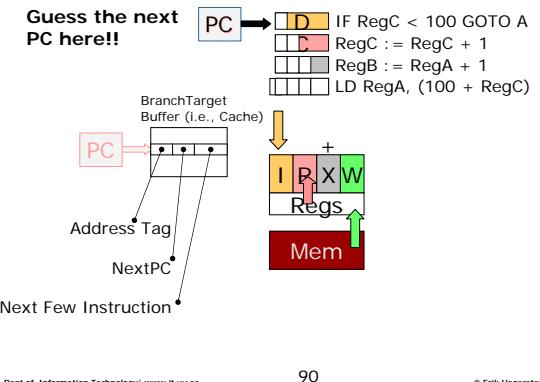
| Mem ref 1       | Mem ref 2       | FP op 1         | FP op 2         | Int op / branch | Clock |
|-----------------|-----------------|-----------------|-----------------|-----------------|-------|
| LD F0,0(R1)     | LD F6,-8(R1)    | NOP             | NOP             | NOP             | 1     |
| LD F10,-16(R1)  | LD F14,-24(R1)  | NOP             | NOP             | NOP             | 2     |
| LD F18,-32(R1)  | LD F22,-40(R1)  | ADDD F4,F0,F2   | ADDD F8,F6,F2   | NOP             | 3     |
| LD F26,-48(R1)  | NOP             | ADDD F12,F10,F2 | ADDD F16,F14,F2 | NOP             | 4     |
| NOP             | NOP             | ADDD F20,F18,F2 | ADDD F24,F22,F2 | NOP             | 5     |
| SD 0(R1), F4    | SD -8(R1), F8   | ADDD F28,F26,F2 | NOP             | NOP             | 6     |
| SD -16(R1), F12 | SD -24(R1), F8  | NOP             | NOP             | NOP             | 7     |
| SD -32(R1), F20 | SD -40(R1), F24 | NOP             | NOP             | SUBI R1,R1,#48  | 8     |
| SD 0(R1), F28   | NOP             | NOP             | NOP             | BNEZ R1,LOOP    | 9     |

VLIW will be revisited later on....

## Predict next PC

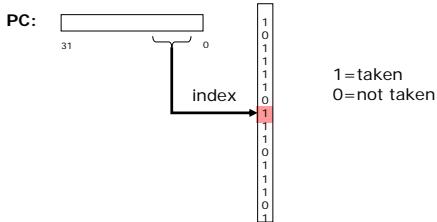


## Cycle 4



## Branch history table

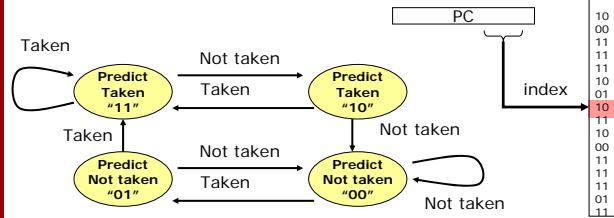
A simple branch prediction scheme



- The branch-prediction buffer is indexed by bits from branch-instruction PC values
- If prediction is wrong, then invert prediction

Problem: can cause two mispredictions in a row

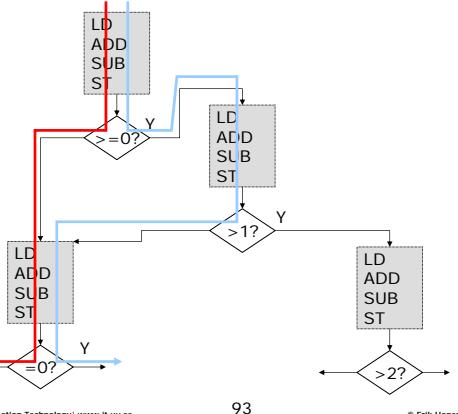
## A two-bit prediction scheme



- Requires prediction to miss twice in order to change prediction => better performance

[Go back to 68](#)

## Dynamic Scheduling Of Branches



## N-level history

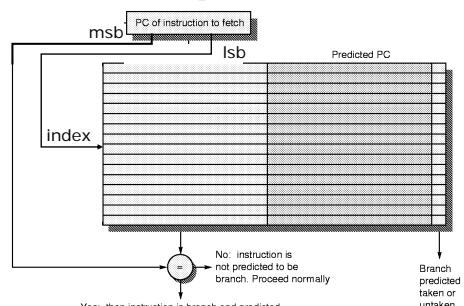
- Not only the PC of the BR instruction matters, also how you've got there is important
- Approach:
  - Record the outcome of the last N branches in a vector of N bits
  - Include the bits in the indexing of the branch table
- Pros/Cons: Same BR instruction may have multiple entries in the branch table

(N,M) prediction = N levels of M-bit prediction

## Tournament prediction

- Issues:
  - No one predictor suits all applications
- Approach:
  - Implement several predictors and dynamically select the most appropriate one
- Performance example SPEC98:
  - 2-bit prediction: 7% miss prediction
  - (2,2) 2-level, 2-bit: 4% miss prediction
  - Tournaments: 3% miss prediction

## Branch target buffer



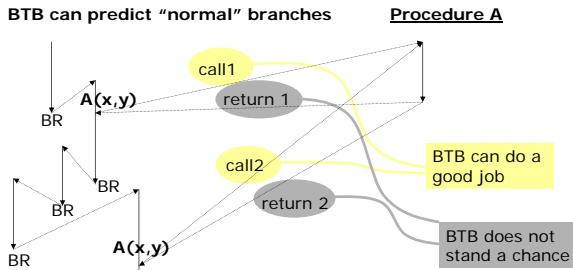
## Putting it together

- BTB stores info about taken instructions
- Combined with a separate branch history table
- Instruction fetch stage highly integrated for branch optimizations

## Folding branches

- BTB often contains the next few instructions at the destination address
- Unconditional branches (and some cond as well) branches execute in zero cycles
  - Execute the dest instruction instead of the branch (*if there is a hit in the BTB at the IF stage*)
  - "Branch folding"

## Procedure calls & BTB



## Return address stack

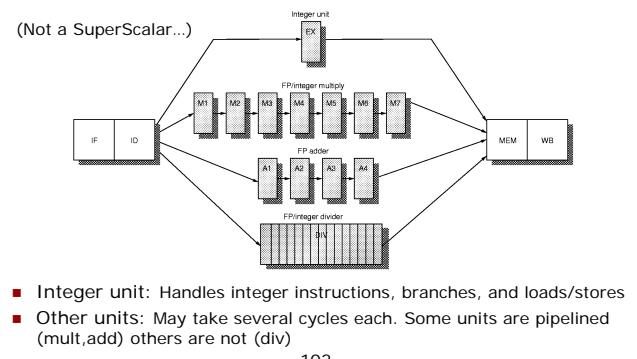
- Popular subroutines are called from many places in the code.
- Branch prediction may be confused!!
- May hurt other predictions
- New approach:
  - Push the return address on a [small] stack at the time of the call
  - Pop addresses on return



## Overlapping Execution

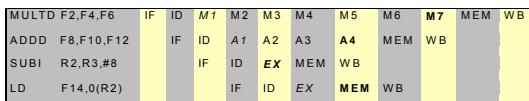
Erik Hagersten  
Uppsala University  
Sweden

## Multicycle operations in the pipeline (floating point)



- Integer unit: Handles integer instructions, branches, and loads/stores
- Other units: May take several cycles each. Some units are pipelined (mult,add) others are not (div)

## Parallelism between integer and FP instructions



How to avoid structural and RAW hazards:

Stall in ID stage when

- The functional unit can be occupied
- Many instructions can reach the WB stage at the same time

RAW hazards:

- Normal bypassing from MEM and WB stages
- Stall in ID stage if any of the source operands is a destination operand of an instruction in any of the FP functional units

AVDARK  
2009

Dept of Information Technology | www.it.uu.se

103

© Erik Hagersten | user.it.uu.se/~eh

## WAR and WAW hazards for multicycle operations

WAR hazards are a non-issue because operands are read in program order (in-order)

WAW hazards are avoided by:

- stalling the SUBF until DIVF reaches the MEM stage, or
- disabling the write to register F0 for the DIVF instruction

WAW Example:

DIVF F0,F2,F4 FP divide 24 cycles

...

SUBF F0,F8,F10 FP sub 3 cycles

SUB finishes before DIV ; out-of-order completion

AVDARK  
2009

Dept of Information Technology | www.it.uu.se

104

© Erik Hagersten | user.it.uu.se/~eh

## Dynamic Instruction Scheduling

Key idea: allow subsequent independent instructions to proceed

DIVD F0,F2,F4 ; takes long time

ADDD F10,F0,F8 ; stalls waiting for F0

SUBD F12,F8,F13 ; Let this instr. bypass the ADDD

- Enables out-of-order execution (& out-of-order completion)

AVDARK  
2009

Two historical schemes used in "recent" machines:

**Tomasulo** in IBM 360/91 in 1967 (also in Power-2)

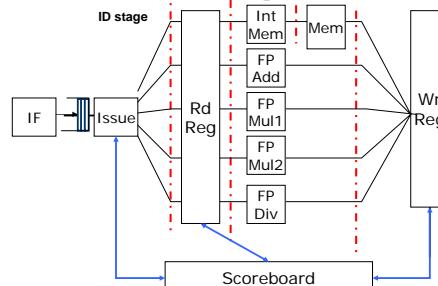
**Scoreboard** dates back to CDC 6600 in 1963

Dept of Information Technology | www.it.uu.se

105

© Erik Hagersten | user.it.uu.se/~eh

## Simple Scoreboard Pipeline (covered briefly in this course)

AVDARK  
2009

Dept of Information Technology | www.it.uu.se

106

© Erik Hagersten | user.it.uu.se/~eh

## Extended Scoreboard

**Issue:** Instruction is issued when:

- No structural hazard for a functional unit
- No WAW with an instruction in execution

**Read:** Instruction reads operands when they become available (RAW)

**EX:** Normal execution

**Write:** Instruction writes when all previous instructions have read or written this operand (WAW, WAR)

AVDARK  
2009

Dept of Information Technology | www.it.uu.se

107

© Erik Hagersten | user.it.uu.se/~eh

## Limitations with scoreboards

The scoreboard technique is limited by:

- Number of scoreboard entries (*window size*)
- Number and types of functional units
- Number of ports to the register bank
- Hazards caused by name dependencies

Tomasulo's algorithm addresses the last two limitations

AVDARK  
2009

Dept of Information Technology | www.it.uu.se

108

© Erik Hagersten | user.it.uu.se/~eh

## A more complicated example

```

DIV    F0,F2,F4      ;delayed a long time
ADD F6,F0,F8
SUBD F6,F10,F14    RAW
SUBD F6,F10,F14    RAW
MULD F6,F10,F8
  
```

WAR and WAW avoided through "register renaming"

Register Renaming:

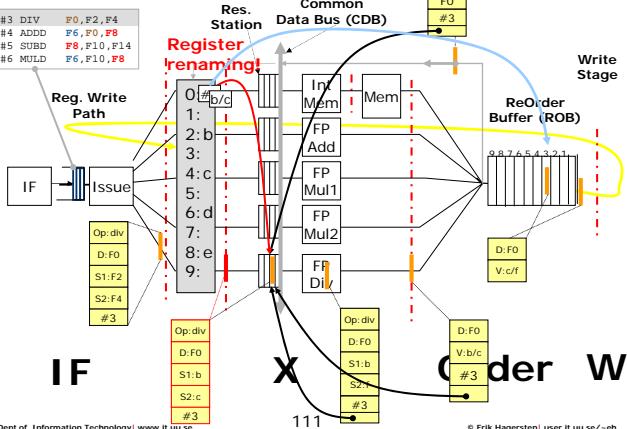
```

DIV    F0,F2,F4
ADD F6,F0,F8
SUBD tmp1,F10,F14 ;can be executed right away
MULD tmp2,F10,tmp1 ;delayed a few cycles
  
```

## Tomasulo's Algorithm

- IBM 360/91 mid 60's
- High performance without compiler support
- Extended for modern architectures
- Many implementations (PowerPC, Pentium...)

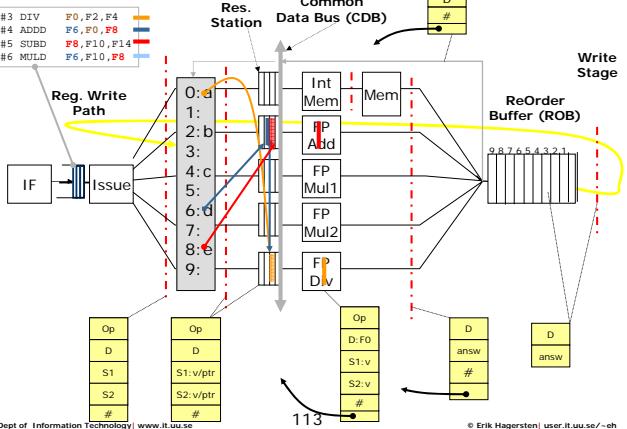
## Simple Tomasulo's Algorithm



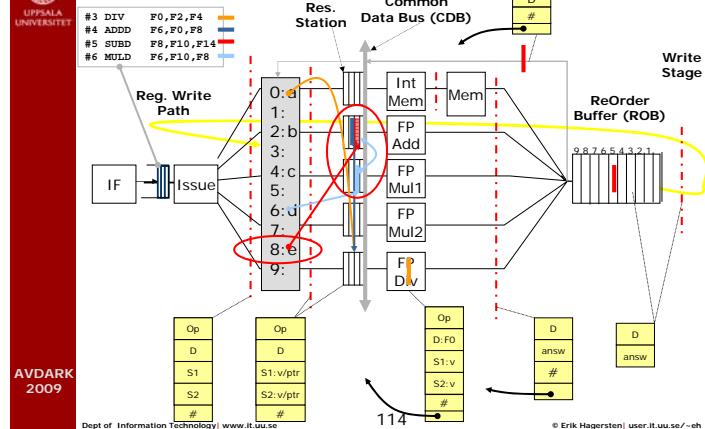
## Tomasulo's: What is going on?

1. Read Register:
  - Rename DestReg to the Res. Station location
2. Wait for all dependencies at Res. Station
3. After Execution
  - a) Put result in Reorder Buffer (ROB)
  - b) Broadcast result on CDB to all waiting instructions
  - c) Rename DestReg to the ROB location
4. When all preceding instr. have arrived at ROB:
  - Write value to DestReg

## Simple Tomasulo's Algorithm

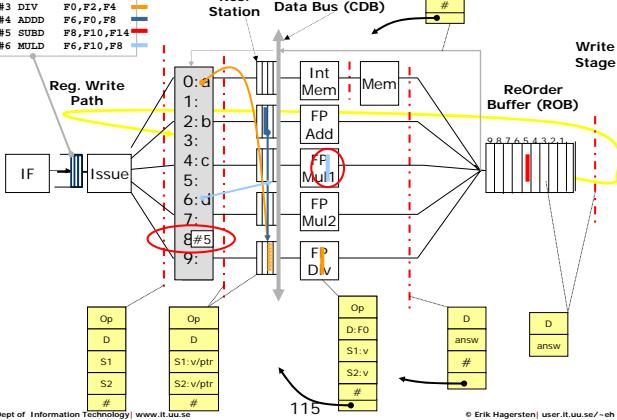


## Simple Tomasulo's Algorithm





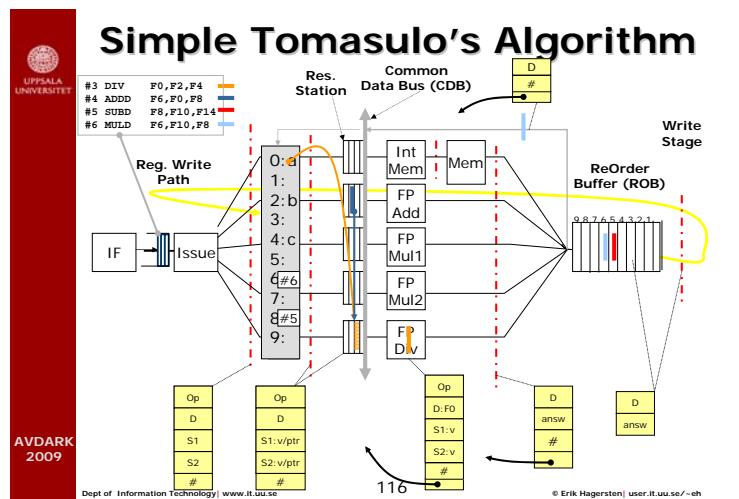
## Simple Tomasulo's Algorithm



AVDAR  
8888

# #  
Dept of Information Technology | www.it.uu.se

© Erik Hagersten | user.it.uu.se/~eh

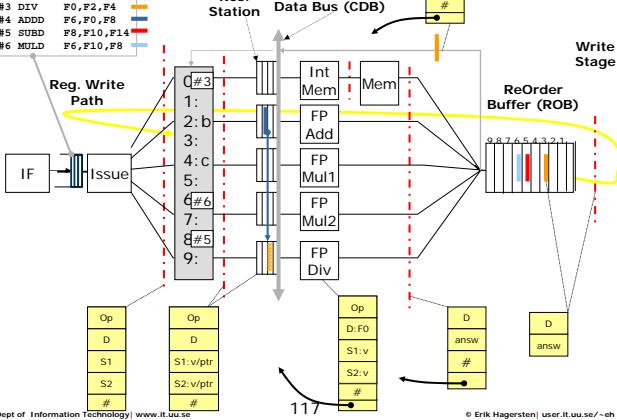


Dept of Information Technology | www

© Erik Hagersten | user.it.uu.se/~eh



# Simple Tomasulo's Algorithm

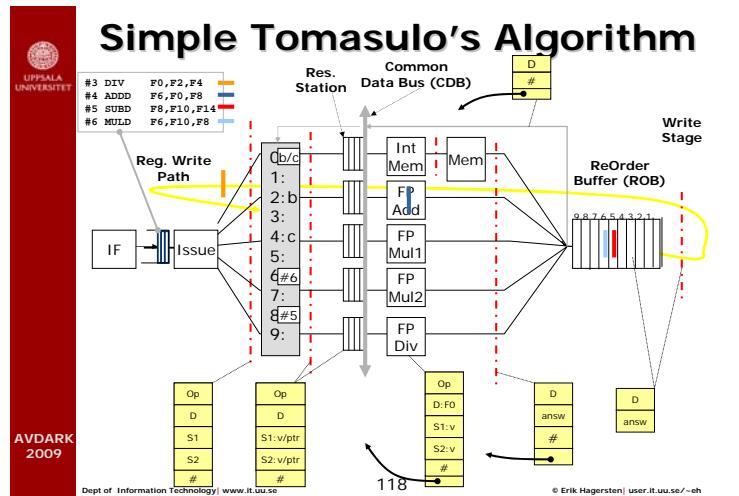


AVDAR

S2: vA

1

© Erik Hagersten | user.it.uu.se/~eh



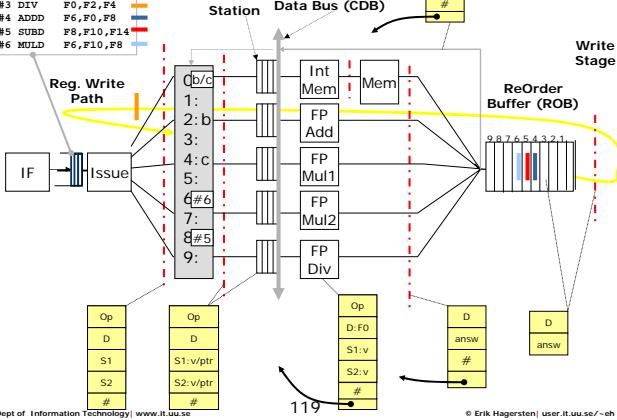
S2

1

© Erik Hagersten | user.it.uu.se/~eh



# Simple Tomasulo's Algorithm

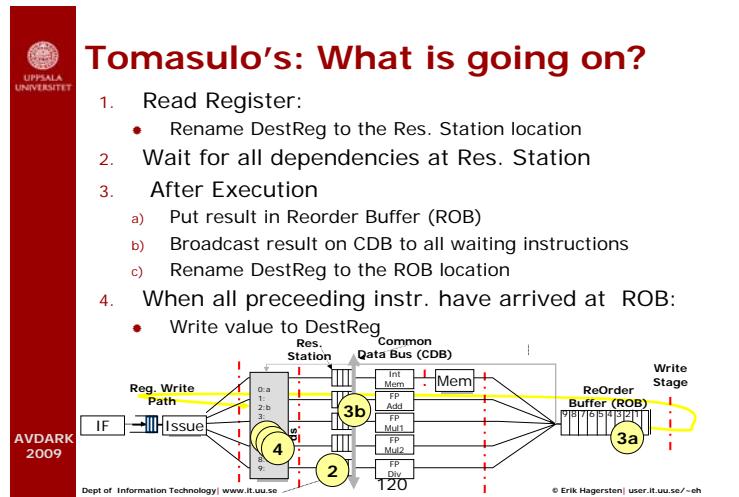


AVDAR

S2 S2:v

Page 1

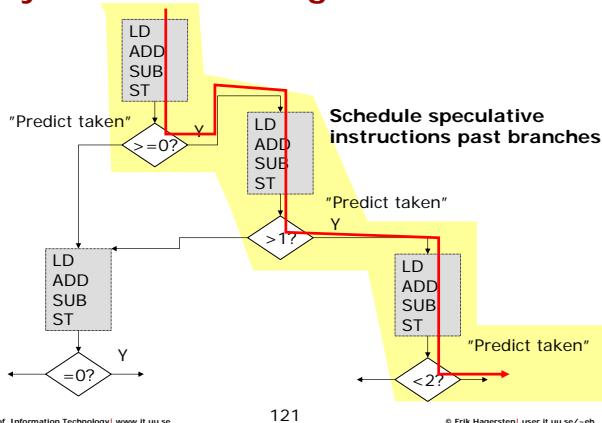
© Erik Hagersten | user.it.uu.se/~eh



2009

1. Read Register:
    - Rename DestReg to the Res. Station location
  2. Wait for all dependencies at Res. Station
  3. After Execution
    - a) Put result in Reorder Buffer (ROB)
    - b) Broadcast result on CDB to all waiting instructions
    - c) Rename DestReg to the ROB location
  4. When all preceding instr. have arrived at ROB:
    - Write value to DestReg

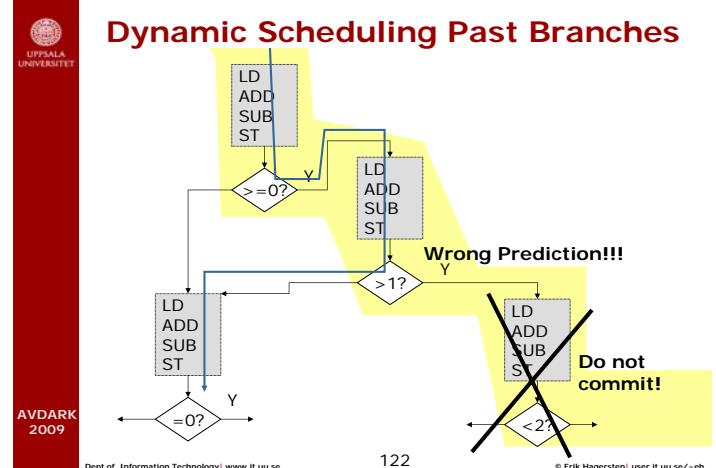
## Dynamic Scheduling Past Branches



121

© Erik Hagersten | user.it.uu.se/~eh

## Dynamic Scheduling Past Branches



122

© Erik Hagersten | user.it.uu.se/~eh

## Summing up Tomasulo's

- Out-of-order (O-O-O) execution
- In order commit
  - Allows for speculative execution (beyond branches)
  - Allows for precise exceptions
- Distributed implementation
  - Reservation stations – wait for RAW resolution
  - Reorder Buffer (ROB)
  - Common Data Bus "snoops" (CDB)
- "Register renaming" avoids WAW, WAR
- Costly to implement (complexity and power)

Dept of Information Technology | www.it.uu.se

123

© Erik Hagersten | user.it.uu.se/~eh

## Dealing with Exceptions

Erik Hagersten  
Uppsala University  
Sweden



## Exception handling in pipelines

Example: Page fault from TLB

Must restart the instruction that causes an exception (interrupt, trap, fault) "precise interrupts"

(...as well as all instructions following it.)

A solution (in-order...):

1. Force a trap instruction into the pipeline
2. Turn off all writes for the faulting instruction
3. Save the PC for the faulting instruction  
- to be used in return from exception

Dept of Information Technology | www.it.uu.se

125

© Erik Hagersten | user.it.uu.se/~eh

## Guaranteeing the execution order

Exceptions may be generated in another order than the instruction execution order

| Pipeline stage | Problem causing exception  |
|----------------|--|
| IF             | Page fault on instruction fetch; misaligned memory access; memory protection violation |
| ID             | Undefined or illegal opcode  |
| EX             | Arithmetic exception   |
| MEM            | Page fault on data access; misaligned memory access; memory protection violation       |
| WB             | none   |

Example sequence:

Iw (e.g., page fault in MEM)  
add (e.g., page fault in IF)

## FP Exceptions

Example: DIVF F0,F2,F4      24 cycles  
ADD F10,F10,F8      3 cycles  
SUBF F12,F12,F14      3 cycles

SUBF may generate a trap before DIVF has completed!!

## Revisiting Exceptions:

A pipeline implements precise interrupts iff:

All instructions before the faulting instruction can complete

All instructions after (and including) the faulting instruction must not change the system state and must be restartable

ROB helps the implementation in O-O-O execution

## HW support for [static] speculation and improved ILP

Erik Hagersten  
Uppsala University  
Sweden

## Very Long Instruction Word (VLIW)

- Independent functional units with no hazard detection

Compiler is responsible for instruction scheduling

| Mem ref 1       | Mem ref 2       | FP op 1         | FP op 2         | Int op/branch  | Clock |
|-----------------|-----------------|-----------------|-----------------|----------------|-------|
| LD F0,0(R1)     | LD F6,-8(R1)    | NOP             | NOP             | NOP            | 1     |
| LD F10,-16(R1)  | LD F14,-24(R1)  | NOP             | NOP             | NOP            | 2     |
| LD F18,-32(R1)  | LD F22,-40(R1)  | ADDD F4,F0,F2   | ADDD F8,F6,F2   | NOP            | 3     |
| LD F26,-48(R1)  | NOP             | ADDD F12,F10,F2 | ADDD F16,F14,F2 | NOP            | 4     |
| NOP             | NOP             | ADDD F20,F18,F2 | ADDD F24,F22,F2 | NOP            | 5     |
| SD 0(R1), F4    | SD -8(R1), F8   | ADDD F28,F26,F2 | NOP             | NOP            | 6     |
| SD -16(R1), F12 | SD -24(R1), F8  | NOP             | NOP             | NOP            | 7     |
| SD -32(R1), F20 | SD -40(R1), F24 | NOP             | NOP             | SUBI R1,R1,#48 | 8     |
| SD 0(R1), F28   | NOP             | NOP             | NOP             | BNEZ R1,LOOP   | 9     |

## Limits to VLIW

Difficult to exploit parallelism

- $N$  functional units and  $K$  "dependent" pipeline stages implies  $N \times K$  independent instructions to avoid stalls

Memory and register bandwidth

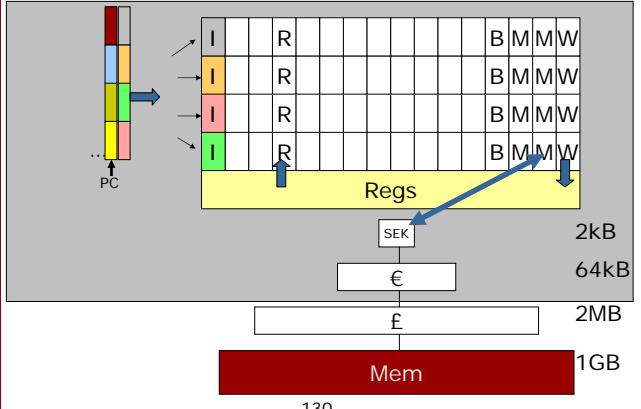
Code size

No binary code compatibility

But, .... simpler hardware

- short schedule
- high frequency

## VLIW: Very Long Instruction Word



## HW support for static speculation

- Move LD up and ST down. But, how far?
  - Normally not outside of the basic block!
- These techniques will allow larger moves and increase the effective size of a basic block
  - Removing branches: predicate execution
  - Move LD above ST: hazard detection
  - Move LD above branch: avoid false exceptions

Dept of Information Technology | www.it.uu.se

133

© Erik Hagersten | user.it.uu.se/~eh

## Compiler speculation

The compiler moves instructions before a branch so that they can be executed before the branch condition is known

Advantage: creates longer schedulable code sequences => more ILP can be exploited

Example: if ( $A == 0$ ) then  $A = B$ ; else  $A = A+4$ ;

|      | <i>Non speculative code</i> | <i>Speculative code</i> |
|------|-----------------------------|-------------------------|
| LW   | R1,0(R3)                    | LW R1,0(R3)             |
| BNEZ | R1,L1                       | + reg rename            |
| LW   | R1,0(R2)                    | LW R14,0(R2)            |
| J    | L2                          | BEQZ R1,L3              |
| L1:  | ADD R1,R1,4                 | ADD R14,R14,4           |
| L2:  | SW 0(R3),R1                 | SW 0(R3),R14            |

- What about exceptions?

Dept of Information Technology | www.it.uu.se

134

© Erik Hagersten | user.it.uu.se/~eh

## Speculative instructions

Moving a LD up, may make it *speculative*

- Moving past a branch
- Moving past a ST (that may be to the same address)

Issues:

- Non-intrusive
- Correct exception handling (again)
- Low overhead
- Good prediction

Dept of Information Technology | www.it.uu.se

135

© Erik Hagersten | user.it.uu.se/~eh

## Example: Moving LD above a branch

LD.s R1, 100(R2) ; "Speculative LD" to R1

.... ; set "poison bit" in R1 if exception

BRNZ R7, #200

...

LD.chk R1 ; Get exception if poison bit of R1 is set

Good performance if the branch is not taken

Dept of Information Technology | www.it.uu.se

136

© Erik Hagersten | user.it.uu.se/~eh

## Example: Moving LD above a ST

LD.a R1, 100(R2) ; "advanced LD"  
; create entry in the ALAT <addr,reg>

....  
ST R7, 50(R3) ; invalidate entry if ALAT addr match

...  
LD.c R1 ; Redo LD if entry in ALAT invalid  
; remove entry in ALAT

ALAT (advanced load address table) is an associative data structure storing tuples of: <addr, dest-reg>

Dept of Information Technology | www.it.uu.se

137

© Erik Hagersten | user.it.uu.se/~eh

## Conditional execution

- Removes the need for some branches ☺
- Conditional Instructions
  - Conditional register move  
CMOVZ R1, R2, R3 ;move R2 to R1 if ( $R3 == 0$ )
  - Compare-and-swap (atomic memory operations later)  
CAS R1, R2, R3 ;swap R2 and mem(R1) if ( $mem(R1) == R3$ )
  - Avoiding a branch makes the basic block larger!!!  
→ More instructions for the code scheduler to play with
- Predicate execution
  - A more generalized technique
  - Each instruction executed if the associated 1-bit predicate REG is 1.

Dept of Information Technology | www.it.uu.se

138

© Erik Hagersten | user.it.uu.se/~eh

## Predicate example

```
IF R1 > R2 then
    LD R7, 100(R1)
    ADD R1, R1, #1
else
    LD R7, 100(R2)
    ADD R2, R2, #1
end
```

Standard Technique

```
CGT R3,R1,R2
BRNZ R3, else
LD R7, 100(R1)
ADD R1, R1, #1
BR end
else: LD R7, 100(R2)
ADD R2, R2, #1
end:
```

5 instr executed in "then path"  
2 branches

AVDARK

2009

Dept of Information Technology | www.it.uu.se

139

© Erik Hagersten | user.it.uu.se/~eh

## Predicate example

```
IF R1 > R2 then
    LD R7, 100(R1)
    ADD R1, R1, #1
else
    LD R7, 100(R2)
    ADD R2, R2, #1
end
```

Using Predicates

```
... {IF R1 > R2 then P6=1;P7=0
      else P6=0;P7=1} ; //one instr!
P6: LD R7, 100(R1)
P6: ADD R1, R1, #1
P7: LD R7, 100(R2)
P7: ADD R2, R2, #1
```

Standard Technique

```
CGT R3,R1,R2
BRNZ R3, else
LD R7, 100(R1)
ADD R1, R1, #1
BR end
else: LD R7, 100(R2)
ADD R2, R2, #1
end:
```

5 instr executed in "then path"  
2 branches

AVDARK

2009

Dept of Information Technology | www.it.uu.se

140

© Erik Hagersten | user.it.uu.se/~eh

## HW vs. SW speculation

Advantages:

- Dynamic runtime disambiguation of memory addresses
- Dynamic branch prediction is often better than static which limits the performance of SW speculation.
- HW speculation can maintain a precise exception model

Main disadvantage:

- Complex implementation and extensive need of hardware resources (conforms with technology trends)

AVDARK

2009

Dept of Information Technology | www.it.uu.se

141

© Erik Hagersten | user.it.uu.se/~eh

## Example:

### IA64 and Itanium(I)

Erik Hagersten  
Uppsala University  
Sweden



## Little of everything

- VLIW
- Advanced loads supported by ALAT
- Load speculation supported by predication
- Dynamic branch prediction
- "All the tricks in the book"

AVDARK

2009

Dept of Information Technology | www.it.uu.se

143

© Erik Hagersten | user.it.uu.se/~eh

## Itanium instructions

| Type  | Instr 1 | Instr 2 | Instr 3 |  |
|---|---------|---------|---------|--|
| 128   |         |         | 41      |  |
| <ul style="list-style-type: none"> <li>■ Instruction bundle (128 bits)           <ul style="list-style-type: none"> <li>• (5bits) template (identifies I types and dependencies)</li> <li>• 3 x (41bits) instruction</li> </ul> </li> </ul> |         |         |         |  |

- Can issue up to two bundles per cycle (6 instr)
- The "Type" specifies if the instr. are independent
- Latencies:

| Instruction     | Latency |
|-----------------|---------|
| I-LD            | 1       |
| FP-LD           | 9       |
| Pred branch     | 0-3     |
| Misspred branch | 0-9     |
| I-ALU           | 0       |
| FP-ALU          | 4       |

Dept of Information Technology | www.it.uu.se

144

© Erik Hagersten | user.it.uu.se/~eh

## Itanium Registers

- 128 65-bit GPR (w/ poison bit)
- 128 82-bit FP REGS
- 64 1-bit predicate REGS
- A bunch of CSRs (control/status registers)

AVDARK  
2009

Dept of Information Technology | www.it.uu.se

145

© Erik Hagersten | user.it.uu.se/~eh

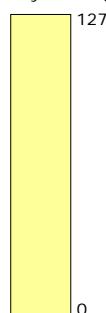
## Dynamic register window

AVDARK  
2009

Dept of Information Technology | www.it.uu.se

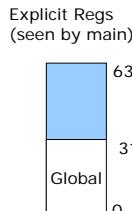
146

Physical Regs



© Erik Hagersten | user.it.uu.se/~eh

## Dynamic register window for GPRs

AVDARK  
2009

147

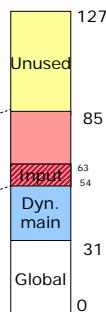
© Erik Hagersten | user.it.uu.se/~eh

AVDARK  
2009

Dept of Information Technology | www.it.uu.se

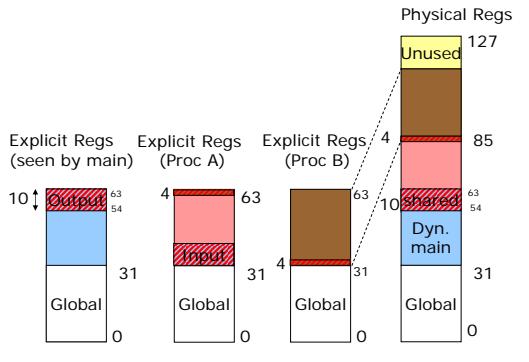
148

Physical Regs



© Erik Hagersten | user.it.uu.se/~eh

## Calling Procedure B (automatic passing of parameters)

AVDARK  
2009

149

© Erik Hagersten | user.it.uu.se/~eh

AVDARK  
2009

## Register Stack Engine (RSE)

- Saves and restores registers to memory on register spills
- Implemented in hardware
- Works in the background
- Gives the illusion of an unlimited register stack
- This is similar to SPARC and UCB's RISC

150

© Erik Hagersten | user.it.uu.se/~eh

## Register rotation: FP and GPRs

- Used in software pipelining
- Register renaming for each iteration
- Removes the need for prologue/epilogue
- RSE (register stack engine)

AVDARK  
2009

## What is the alternative?

- VLIW was meant to simplify HW
- Itanium I has 230 M transistors and consumes 130W?
- Will it scale with technology?
- Other alternatives:
  - Increase cache size,
  - Increase the frequency, or,
  - Run more than one thread/chip (More about this during "Future Technologies")

AVDARK  
2009