MO401

IC/Unicamp

Prof Mario Côrtes

Capítulo 3 – Parte A (3.1 – 3.7):

Instruction-Level Parallelism and Its Exploitation

Tópicos - estrutura

- Parte A
  - Basic compiler ILP
  - Advanced branch prediction
  - Dynamic scheduling
  - Hardware based speculation
  - Multiple issue and static scheduling
- Parte B
  - Instruction delivery and speculation
  - Limitations of ILP
  - ILP and memory issues
  - Multithreading

IC-UNICAMP

3.1 Introduction

- Pipelining become universal technique in 1985
  - Overlaps execution of instructions
  - Exploits “Instruction Level Parallelism”

- Beyond this, there are two main approaches:
  - Hardware-based dynamic approaches
    - Used in server and desktop processors
    - Not used as extensively in PMD processors
  - Compiler-based static approaches
    - Not as successful outside of scientific applications (ruim para inteiros)
    - (menor consumo de energia → era mais popular em PMDs → mas está mudando)

Instruction-Level Parallelism

- When exploiting instruction-level parallelism, goal is to maximize CPI
  - Pipeline CPI = Ideal pipeline CPI + Structural stalls + Data hazard stalls + Control stalls

- What is ILP?
  - Basic block: trecho de código sem branches (só no início e no final)
  - Parallelism with basic block is limited
    - Typical size of basic block = 3-6 instructions
    - Pequenos basic blocks → provável que instruções tenham dependências
    - Must optimize across branches

IC-UNICAMP

Efeito das técnicas sobre ILP

Technique	Reduces	Section
Forwarding and bypassing	Potential data hazard stalls	C.2
Delayed branches and simple branch scheduling	Control hazard stalls	C.2
Basic compiler pipeline scheduling	Data hazard stalls	C.2, 3.2
Basic dynamic scheduling (scoreboarding)	Data hazard stalls from true dependences	C.7
Loop unrolling	Control hazard stalls	3.2
Branch prediction	Control stalls	3.3
Dynamic scheduling with renaming	Stalls from data hazards, output dependences, and antidependences	3.4
Hardware speculation	Data hazard and control hazard stalls	3.6
Dynamic memory disambiguation	Data hazard stalls with memory	3.6
Issuing multiple instructions per cycle	Ideal CPI	3.7, 3.8
Compiler dependence analysis, software pipelining, trace scheduling	Ideal CPI, data hazard stalls	H.2, H.3
Hardware support for compiler speculation	Ideal CPI, data hazard stalls, branch hazard stalls	H.4, H.5

Figure 3.1 The major techniques examined in Appendix C, Chapter 3, and Appendix H are shown together with the component of the CPI equation that the technique affects.

Data Dependence


```
for (i=0; i<=999; i=i+1)
        x[i] = x[i] + y[i];
```

- Loop-Level Parallelism
  - Há loops cujas iterações são independentes
  - Unroll loop statically or dynamically
    - diminui penalidade branch e dependência de dados com register renaming
  - Use SIMD (vector processors and GPUs)
    - data level parallelism (Ch 4); no exemplo acima, 7 instruções / iteração * 1000= 7000 instruções; Se SIMD executa 4 dados dos vetores por vez → speedup =4
- Dependences:
  - control dependences
  - data (true) dependences, name dependences
- Data dependency
  - Instruction *j* is data dependent on instruction *i* if
    - Instruction *i* produces a result that may be used by instruction *j*
    - Instruction *j* is data dependent on instruction *k* and instruction *k* is data dependent on instruction *i*
- Dependent instructions cannot be executed simultaneously

IC-UNICAMP

Data Dependence

- Dependencies are a property of programs
- Pipeline organization determines if dependence is detected and if it causes a stall
- Dependência pode ser tratada de 2 formas:
  - manter dependência mas evitar o hazard
  - modificar o código e eliminar a dependência
- Data dependence conveys:
  - Possibility of a hazard
  - Order in which results must be calculated
  - Upper bound on exploitable instruction level parallelism
- Dependencies that flow through memory locations are difficult to detect
  - 20(R4) e 100(R6) podem apontar para a mesma posição de mem.
  - 20(R4) e 20(R4) podem apontar posições diferentes (instantes dif.)

IC-UNICAMP

Name Dependence

- Two instructions use the same name but no flow of information (two types: WAR and WAW)
  - Not a true data dependence, *but is a problem when reordering instructions*
  - *Antidependence*: instruction j writes a register or memory location that instruction i reads (WAR)
    - Initial ordering (i before j) must be preserved
    - exemplo

```
S.D     F4,  0(R1)
DADDIU  R1,  R1, #-8
```

  - *Output dependence*: instruction i and instruction j write the same register or memory location (WAW)
    - Ordering must be preserved
- To resolve, use renaming techniques
  - não é dependência “verdadeira”; não há dados sendo transmitidos entre I e j
  - mais fácil, se operandos são registradores → register renaming

IC-UNICAMP

Other Factors

- Data Hazards
  - Read after write (RAW)
  - Write after write (WAW)
  - Write after read (WAR)

  - e RAR, seria um hazard?

- Control Dependence
  - Ordering of instruction i with respect to a branch instruction
    - Instruction control dependent on a branch cannot be moved before the branch so that its execution is no longer controller by the branch
      - ex: mover uma instrução que está dentro do “then” de um if-then-else para antes do if
    - An instruction not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch
      - ex: mover uma instrução que aparece antes de um if para dentro do “then” do if-then-else

```
if p1 {
      S1;
};
if p2 {
      S2;
}
```

IC-UNICAMP

Other Factors (cont)

- O que interessa preservar
  - Se execução for feita sem mudança na ordem → não há hazard control dependence
  - Mas se for possível alterar ordem sem afetar a correção do programa → não há problema
  - O que interessa preservar
    - Não: control dependence
    - Sim: correção do programa → data flow correto (inclusive sob exceção)

Exemplos

- Ex1:
  - dependência de R2; afeta correção
  - mas e dependências de controle?
    - BEQz e LW poderiam ser invertidos especulativamente
    - mas LW poderia gerar page fault → exceção

```
        DADDU   R2,R3,R4
        BEQZ    R2,L1
        LW      R1,0(R2)
L1:
```

- Ex2:
  - para preservar correção pode não bastar data flow → control também afeta
  - Valor de R1 no OR depende de o BEQZ ter sido tomado ou não
  - OR depende de DSUBU e de DADDU

```
        DADDU   R1,R2,R3
        BEQZ    R4,L
        DSUBU   R1,R5,R6
L:      ...
        OR      R7,R1,R8
```

- Ex3:
  - as vezes é possível violar control dependence sem afetar data flow e exception
  - Supor R4 não está vivo (live) na região de código depois de skip → possível mover DSUBU para antes de BEQZ

```
        DADDU   R1,R2,R3
        BEQZ    R12,skip
        DSUBU   R4,R5,R6
        DADDU   R5,R4,R9
skip:   OR      R7,R8,R9
```

3.2 Compiler Techniques for Exposing ILP

- Pipeline scheduling
  - Separate dependent instruction from the source instruction by the pipeline latency of the source instruction
- Hipóteses: pipeline inteiro de 5 estágios; branches have a delay of one clock cycle; units fully pipelined; no structural hazard
- Example: como transformar o loop → mais ILP disponível?

```
for (i=999; i>=0; i=i-1)            ## iterações independentes
  x[i] = x[i] + s;
```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Loop em assembly do MIPS

- for (i=999; i>=0; i=i-1)
  x[i] = x[i] + s;

```
Loop:   L.D      F0,0(R1)       ## R1 = high array addr
        ADD.D    F4,F0,F2       ## F2 ← scalar s
        S.D      F4,0(R1)
        DADDUI   R1,R1,#-8
        BNE      R1,R2,Loop     ## R2 = stop elem. addr.
```

IC-UNICAMP

Exmpl p158

			ciclo emitido
Loop:	L.D	F0,0(R1)	1
	stall		2
	ADD.D	F4,F0,F2	3
	stall		4
	stall		5
	S.D	F4,0(R1)	6
	DADDUI	R1,R1,#-8	7
	stall		8
	BNE	R1,R2,Loop	9

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Exmpl p158: after scheduling

Scheduled code:

			ciclo emitido
Loop:	L.D	F0,0(R1)	1
	DADDUI	R1,R1,#-8	2
	ADD.D	F4,F0,F2	3
	stall		4
	stall		5
	S.D	F4,8(R1)	6
	BNE	R1,R2,Loop	7

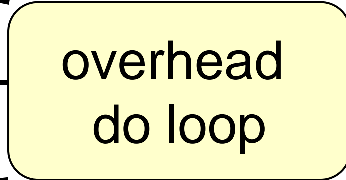
- 7 ciclos, mas
- apenas 3 ciclos úteis: L.D, ADD, S.D
- 4 restantes → loop overhead

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Exmpl p159: Loop Unrolling

- Loop unrolling (on same code); hipótese: array size = múltiplo de 32
  - Unroll by a factor of 4 (assume # elements is divisible by 4)
  - Eliminate unnecessary instructions

```
Loop:   L.D F0,0(R1)
        ADD.D F4,F0,F2
        S.D F4,0(R1) ;drop DADDUI & BNE
------------------------------------
        L.D F6,-8(R1)
        ADD.D F8,F6,F2
        S.D F8,-8(R1) ;drop DADDUI & BNE
------------------------------------
        L.D F10,-16(R1)
        ADD.D F12,F10,F2
        S.D F12,-16(R1) ;drop DADDUI & BNE
------------------------------------
        L.D F14,-24(R1)
        ADD.D F16,F14,F2
        S.D F16,-24(R1)
------------------------------------
        DADDUI R1,R1,#-32
        BNE R1,R2,Loop
```



- note: number of live registers vs. original loop

- Lots of data dependencies (solution ahead)
- Total 27 ciclos → média de 6.75 ciclos / iter

Exmpl p159: Loop Unrolling + scheduling

- Pipeline schedule the unrolled loop:

```
Loop:   L.D F0,0(R1)
        L.D F6,-8(R1)
        L.D F10,-16(R1)
        L.D F14,-24(R1)
        ADD.D F4,F0,F2
        ADD.D F8,F6,F2
        ADD.D F12,F10,F2
        ADD.D F16,F14,F2
        S.D F4,0(R1)
        S.D F8,-8(R1)
        DADDUI R1,R1,#-32
        S.D F12,16(R1)
        S.D F16,8(R1)
        BNE R1,R2,Loop
```

Loads

Adds

Stores (um DADDUI invertido)

- No more data dependencies
- Total 14 ciclos → média de 3.5 ciclos / iter

Strip Mining

- Unknown number of loop iterations?
  - Number of iterations = *n*
  - Goal: make *k* copies of the loop body
  - Generate pair of loops:
    - First executes *n* mod *k* times
    - Second executes *n* / *k* times
    - "Strip mining"

    - (texto nas páginas 310 e 311 do Cap 4 ou Ap G)

IC-UNICAMP

3.3 Branch Prediction

- In Ap C, we saw simple branch prediction techniques
  - In chapter 3, new techniques → more accurate prediction
- Basic 2-bit predictor: (visto no Ap C)
  - For each branch:
    - Predict taken or not taken
    - If the prediction is wrong two consecutive times, change prediction
- Correlating predictor:
  - Multiple 2-bit predictors for each branch
  - One for each possible combination of outcomes of preceding *n* branches
- Local predictor:
  - Multiple 2-bit predictors for each branch
  - One for each possible combination of outcomes for the last *n* occurrences of this branch
- Tournament predictor:
  - Combine correlating predictor with local predictor

Correlating branch predictor


```
if (aa==2)
        aa=0;
if (bb==2)
        bb=0;
if (aa!=bb) {
```

```
        DADDIU   R3,R1,#-2
        BNEZ     R3,L1          ;branch b1    (aa!=2)
        DADD     R1,R0,R0       ;aa=0
L1:     DADDIU   R3,R2,#-2
        BNEZ     R3,L2          ;branch b2    (bb!=2)
        DADD     R2,R0,R0       ;bb=0
L2:     DSUBU    R3,R1,R2       ;R3=aa-bb
        BEQZ     R3,L3          ;branch b3    (aa==bb)
```

- Comportamento do branch b3 está correlacionado com b1 e b2.
  - se b1 e b2 “not taken” → b3 “taken”
- Um esquema que guarde o histórico individual dos branches (uncorrelating branch predictor) não capta este comportamento
- Para isso → correlating predictors ou two-level predictors

Correlating branch predictor (cont)

- Armazena informação sobre os últimos branches para tomar decisão sobre o atual
- ex: (1,2) usa o comportamento do último branch para escolher entre um par de preditores de 2 bits
- ex: (m,n) usa o comportamento dos últimos "m" branches para escolher um branch predictor de n bits dentre 2^m preditores (cada um, para um único branch)
- Vantagens: melhora desempenho e hardware é simples
- Hardware:
  - histórico: shift register de m bits, cada bit registra se aquele branch foi tomado ou não
  - o branch prediction buffer pode ser indexado por:
    (lower order branch address) concat (m-bit shift register)
- Ex: Um buffer (2,2) com 64 linhas
  - índice = 6 bits = 4 bits (branch address) concat 2 bits (shift register)
- Para comparar efetividade, esquemas com mesmo hw
  - bits em (m, n) = 2^m x n x (itens selecionados pelo endereço)

IC-UNICAMP

Correlating branch predictor (cont)

- Para comparar efetividade, esquemas com mesmo hw
  - bits em (m, n) = 2^m x n x (itens selecionados pelo endereço)
- Exemplo anterior: (2,2) com 64 linhas
  - m=2, n=2, endereço de 4 bits → 2^4 = 16;
    2^2 x 2 x 2^4 = 128 (matriz de 64 linhas, 2 bits/linha)
- Exemplo1 página 164
  - (0,2) com 4k linhas? (4k precisam 12 bits de índice)
    → m=0, n=2, 12 bits → 2^0 x 2 x 2^{12} = 8k
- Exemplo2 página 164
  - (2,2) com 8k total de hardware?
    2^2 x 2 x (nº itens selecionados) = 8k
  - 8 x (nº itens selecionados) = 8k
  - nº itens selecionados = 1k → 10 lower bits do endereço

IC-UNICAMP

Desempenho com o mesmo hardware

Figure 3.3 Comparison of 2-bit predictors. A **noncorrelating** predictor for 4096 bits is first, followed by a **noncorrelating** 2-bit predictor with unlimited entries and a 2-bit **correlating** predictor with 2 bits of global history and a total of 1024 entries. Although these data are for an older version of SPEC, data for more recent SPEC benchmarks would show similar differences in accuracy.

SPEC89 benchmarks	4096 entries: 2 bits per entry	Unlimited entries: 2 bits per entry	1024 entries: (2,2)
nASA7	1%	0%	1%
matrix300	0%	0%	0%
tomcatv	1%	0%	1%
doduc	5%	5%	5%
spice	9%	9%	5%
fpppp	9%	9%	5%
gcc	12%	11%	11%
espresso	5%	5%	4%
eqntott	18%	18%	6%
li	10%	10%	5%

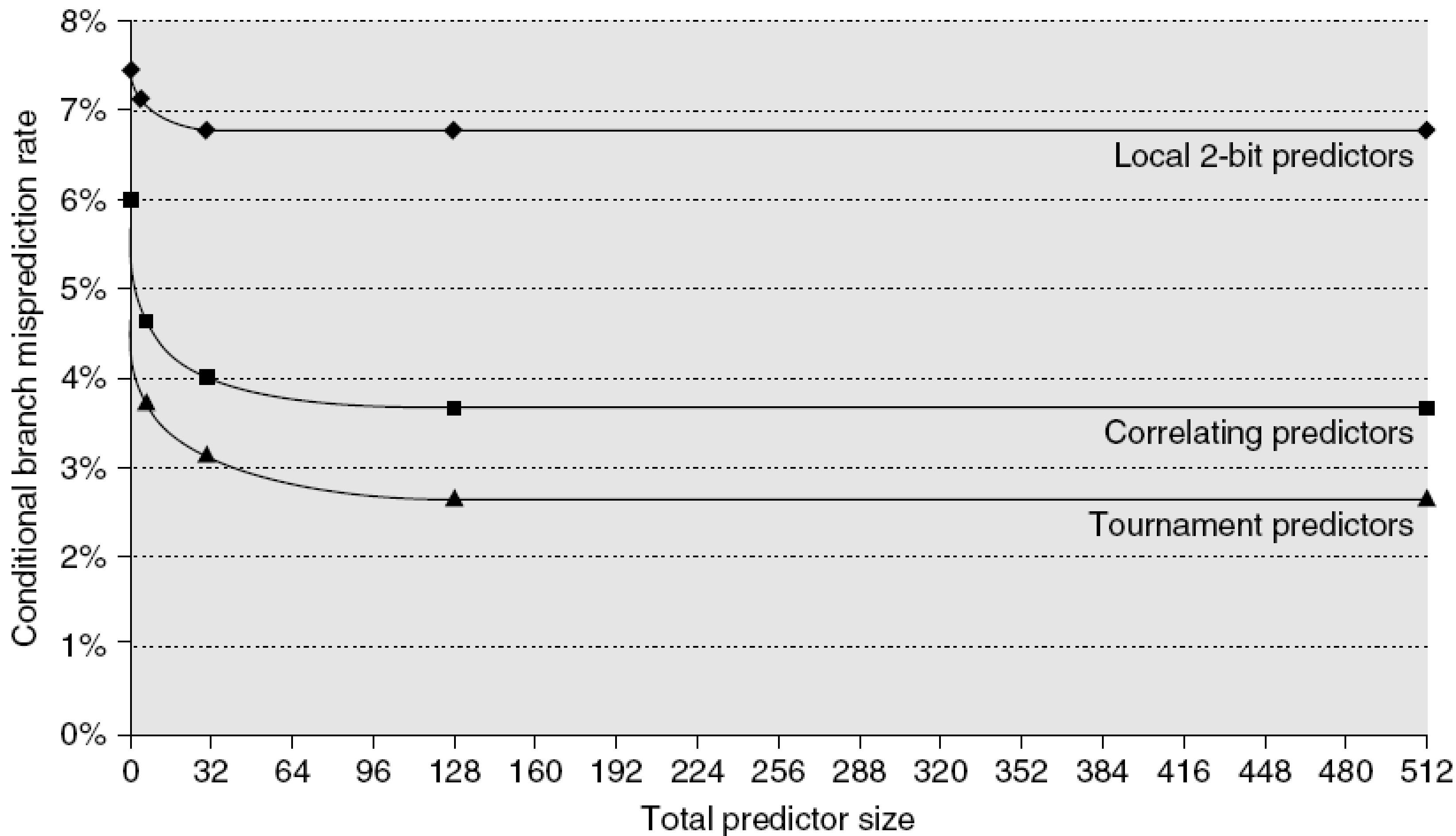
Frequency of mispredictions (0% – 18%)

Tournament predictors

- Múltiplos preditores a serem selecionados
  - informação local
  - informação global
- Ex: 2 bit por branch (como no uncorrelated) escolhem qual preditor a usar: local, global ou misto
- Capacidade de escolher entre local e global → muita diferença para benchmarks inteiros
  - tipicamente:
    - inteiros: escolhe global 40% das vezes
    - fp: escolhe global 15% das vezes

IC-UNICAMP

Branch Prediction Performance



Branch predictor performance

Misprediction rate no Intel i7

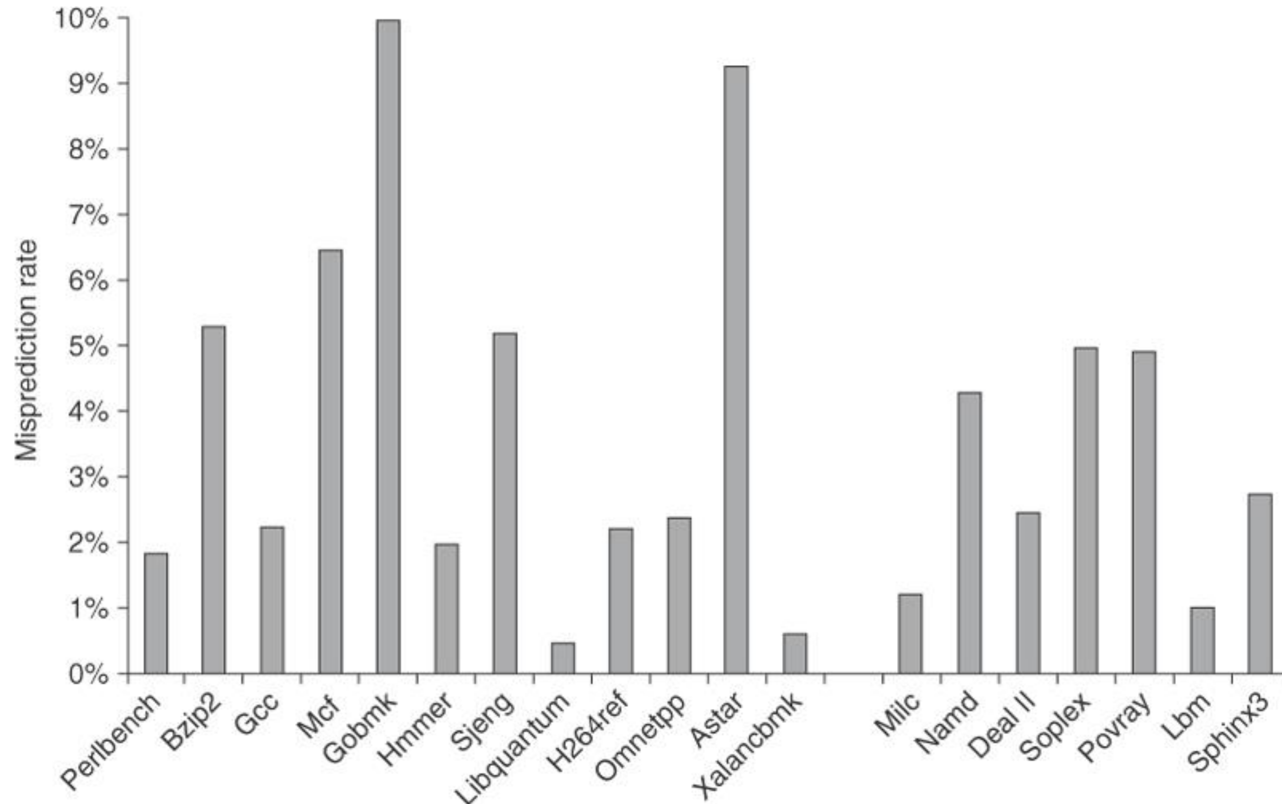


Figure 3.5 The misprediction rate for 19 of the SPECCPU2006 benchmarks versus the number of successfully retired branches is slightly higher on average for the integer benchmarks than for the FP (4% versus 3%). More importantly, it is much higher for a few benchmarks.

3.4 Dynamic Scheduling

- Rearrange order of instructions to reduce stalls while maintaining data flow

- Advantages:
  - Compiler doesn’t need to have knowledge of microarchitecture
  - Handles cases where dependencies are unknown at compile time
    - ex: memory reference, data-dependent branch
  - Permite que o processador tolere latências imprevisíveis
    - ex: após cache miss, pode fazer algo útil em vez de stall

- Disadvantage:
  - Substantial increase in hardware complexity
  - Complicates exceptions

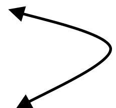
Dynamic Scheduling

- Simple pipeline: in-order issue and execution

  DIV.D **F0**, F2, F4
  ADD.D F10, **F0**, F8 # stall
  SUB.D F12, F8, F14 # poderia ser executada, mas stall

- Para re-ordenar, issue em 2 passos
  - verificar hazards
  - enviar para unidade de execução
- In-order issue and Out-of-order execution
  - Creates the possibility for WAR and WAW hazards
- Ex (outro)

  DIV.D F0, F2, F4
  ADD.D F6, F0, **F8**
  SUB.D **F8**, F10, F14
  MUL.D F6, F10, F8



antidependence

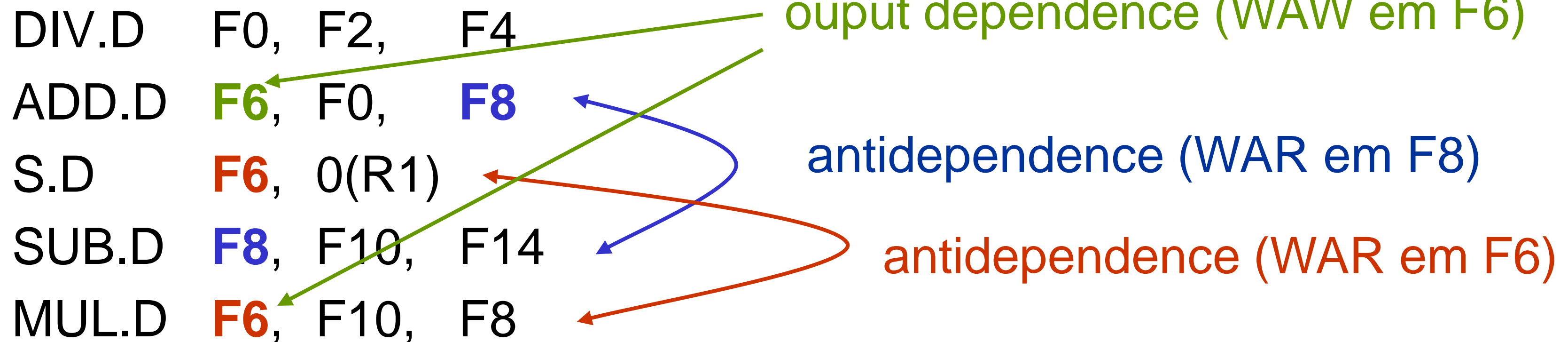
- Se o pipeline inverter ADD.D e SUB.D → erro

Dynamic Scheduling

- Complicações adicionais: precise exception
- Às vezes, não há como evitar imprecise exception
  - pipeline já completou instruções posteriores àquela que gerou a exceção
  - pipeline ainda não completou instrução anterior àquela que gerou a exceção
  - soluções adiante
- Para execução fora de ordem, dividir estágio ID
  - decodificar instrução, verificar hazard
  - aguardar hazards e ler operandos
- Scoreboarding e sua evolução: algoritmo de Tomasulo

Register Renaming – WAR e WAW

- Example (3 hazards):



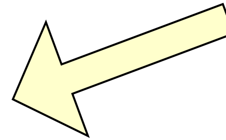
- Além disso há 3 RAW (true data dependencies)
  - F0 (DIV.D e ADD.D)
  - F8 (SUB.D e MUL.D)
  - F6 (ADD.D e S.D)

- As antidependências podem ser eliminadas por register renaming

Register Renaming

- Example:

DIV.D	F0,	F2,	F4
ADD.D	S,	F0,	F8
S.D	S,	0(R1)	
SUB.D	T,	F10,	F14
MUL.D	F6,	F10,	T



DIV.D	F0,	F2,	F4
ADD.D	**F6**,	F0,	**F8**
S.D	**F6**,	0(R1)	
SUB.D	**F8**,	F10,	F14
MUL.D	**F6**,	F10,	F8

- Supor existência de 2 registradores temporários S e T
- Usos posteriores de F8 devem ser substituídos por T
  - pode ser feito estaticamente pelo compilador, mas é complicado
- O algoritmo de Tomasulo trata corretamente renaming entre loops diferentes
- Now only RAW hazards remain, which can be strictly ordered

IC-UNICAMP

Register Renaming no Alg de Tomasulo

- Register renaming is provided by reservation stations (RS)
  - Contains:
    - The instruction
    - Buffered operand values (when available)
    - Number of the reservation station of the instruction that will provide the operand values
  - RS fetches and buffers an operand as soon as it becomes available (not necessarily involving register file)
  - Pending instructions designate the RS to which they will send their output
    - Result values broadcast on a result bus, called the common data bus (CDB)
  - When successive writes to a register, only the last output updates the register file
  - As instructions are issued, the register specifiers to the operands are renamed with the reservation station number where operands are (will be)
  - May be more reservation stations than registers
- Other aspects
  - Hazard detection: agora centralizado (antes, controle e detecção distribuidos)
  - Resultados: direto da RS para unidades funcionais, via CDB (Common Data Bus). Semelhante ao forwarding

IC-UNICAMP

Tomasulo's Algorithm

- Load and store buffers
  - Contain data and addresses, act like reservation stations
- Tom. Algor.
  - arb # cycles
  - 3 steps
    - Issue
    - Execute
    - Write result

IC-UNICAMP

Tomasulo’s 1st step: Issue

- Get next instruction from FIFO queue
- If available RS, issue the instruction to the RS along with operand values (if available) if there is RS available
- If no RS available → stall (structural hazard)
- If operand will be produced by pending instruction, keep track of the functional unit that will produce it

From instruction unit
Instruction queue
FP registers
Load/store operations
Floating-point operations
Operand buses
Address unit
Store buffers
Load buffers
Operation bus
3
2
1
Reservation stations
2
1
Data
Address
Memory unit
FP adders
FP multipliers
Common data bus (CDB)

IC-UNICAMP

Tomasulo's 2nd step: Execute

- When operand becomes available (DCB), store it in any reservation stations waiting for it
- When all operands are ready, issue the instruction → solve RAW
  - If many instructions are ready for a unit, choose one
- Loads and store maintained in program order through effective address
- No instruction allowed to initiate execution until all branches that proceed it in program order have completed →care with precise exception

From instruction unit
Instruction queue
FP registers
Load/store operations
Floating-point operations
Operand buses
Address unit
Store buffers
Load buffers
Operation bus
3
2
1
Reservation stations
2
1
Data
Address
Memory unit
FP adders
FP multipliers
Common data bus (CDB)

IC-UNICAMP

Tomasulo's 3rd step: Write Result

- Write result on CDB into reservation stations and store buffers
  - (Stores must wait until address and value are received)
- From reservation → registers

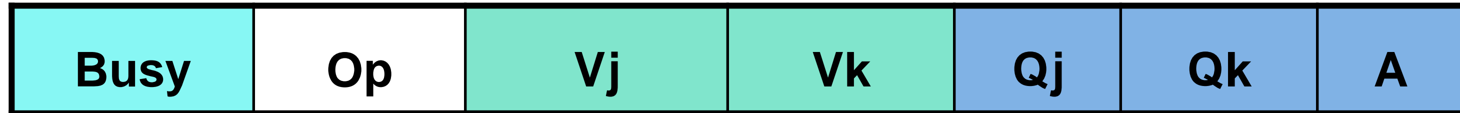
From instruction unit
Instruction queue
FP registers
Load/store operations
Floating-point operations
Operand buses
Address unit
Store buffers
Load buffers
Operation bus
3
2
1
Reservation stations
2
1
Data
Address
Memory unit
FP adders
FP multipliers
Common data bus (CDB)

Tomasulo´s Alg: other aspects

- Data structures associated to RS, RF,LD/STO buffers
- Tags: names used to extend register for renaming purposes
  - indicate which RS will produce needed operand
- Issued instruction → wait source operand → refers to RS where it will be written
- Intermediate results act as renamed registers → solves WAW , WAR
- Results are broadcast (CDB), monitored by RS
  - implements forwarding and bypass
  - but, one extra cycle between producing and consuming result
- Tags refer to buffer or FU that will produce a result
  - register names are discarded as instruction → RS
  - on contrary, in scoreboarding operands stay in the registers → no register renaming

Data structures

- RS



  - Busy: this station and accompanying FU is(not) busy
  - OP: operation to be performed on source operands S1 and S2
  - Vj, Vk: the value of the source operands
    - only one of V fields or the Q field is valid for each operand
    - in loads, Vk holds the offset
  - Qj, Qk: the RSs that will produce source operands; (zero means operand already available or not necessary)
  - A: info for memory address calculation for LD/ST
    - initially, immediate address; then, effective address
- Register File
  - Qi: # of RS that will hold the result to be stored in this register
    - if zero: no one computes results to be stored here (register value can be used)
- LD/ST buffer (each)
  - A: effective address (once 1st step of execution done)

IC-UNICAMP

3.5 Exmpl p176

Instruction		Instruction status		
		Issue	Execute	Write Result
L.D	F6,32(R2)	√	√	√
L.D	F2,44(R3)	√	√	
MUL.D	F0,F2,F4	√		
SUB.D	F8,F2,F6	√		
DIV.D	F10,F0,F6	√		
ADD.D	F6,F8,F2	√		

	Reservation stations						
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	No						
Load2	Yes	Load					44 + Regs[R3]
Add1	Yes	SUB		Mem[32 + Regs[R2]]	Load2		
Add2	Yes	ADD			Add1	Load2	
Add3	No						
Mult1	Yes	MUL		Regs[F4]	Load2		
Mult2	Yes	DIV		Mem[32 + Regs[R2]]	Mult1		

	Register status							
Field	F0	F2	F4	F6	F8	F10	F12	... F30
Qi	Mult1	Load2		Add2	Add1	Mult2		

Figure 3.7 Reservation stations and register tags shown when all of the instructions have issued, but only the first load instruction has completed and written its result to the CDB. The second load has completed effective address calculation but is waiting on the memory unit. We use the array Regs[ ] to refer to the register file and the array Mem[ ] to refer to the memory. Remember that an operand is specified by either a Q field or a V field at any time. Notice that the ADD.D instruction, which has a WAR hazard at the WB stage, has issued and could complete before the DIV.D initiates.

IC-UNICAMP

Exemplo do Alg. Tomasulo (p176)

- Trecho de programa a ser executado:

```
1   L.D     F6,   34(R2)
2   L.D     F2,   45(R3)
3   MUL.D   F0,   F2,   F4
4   SUB.D   F8,   F2,   F6
5   DIV.D   F10,  F0,   F6
6   ADD.D   F6,   F8,   F2
```

RAW?: (1-4); (1-5); (2-3); (2-4); (2-6); ....
WAW?: (1-6)
WAR?: (5-6)

Exemplo do Alg. Tomasulo

- Assumir as seguintes latências:
  - Load: 1 ciclo
  - Add; 2 ciclos
  - Multiplicação: 10 ciclos
  - Divisão: 40 ciclos
- Load-Store:
  - Calcula o endereço efetivo (FU)
  - Load ou Store buffers
  - Acesso à memória (somente load)
  - Write Result
    - Load: envia o valor para o registador e/ou reservation stations
    - Store: escreve o valor na memória
    - (escritas somente no estágio "WB" – simplifica o algoritmo de Tomasulo)

Exemplo do Alg. Tomasulo

IC-UNICAMP

Instruções do programa

3 estágios da execução

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2			
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

3 Load/Buffers

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Latência (que falta) da FU

3 FP Adder R.S.
2 FP Mult R.S.

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
0	*FU*									

Clock cycle

IC-UNICAMP

Exemplo Tomasulo: Ciclo 1

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1		
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
1	FU				Load1					

IC-UNICAMP

Exemplo Tomasulo: Ciclo 2

Instruction status:

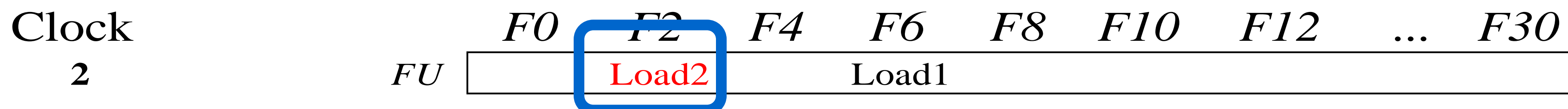
Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1		
LD	F2	45+	R3	2		
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:



Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
2	*FU*		Load2		Load1					

Nota: pode haver múltiplos loads pendentes

Exemplo Tomasulo: Ciclo 3

IC-UNICAMP

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	
LD	F2	45+	R3	2		
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F4)	Load2	
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
3	*FU*	Mult1	Load2		Load1					

- **Nota: nomes dos registradores são removidos ("renamed") na Reservation Stations; MULT issued**
- **Load1 completa; alguém esperando por Load1?**

IC-UNICAMP

Exemplo Tomasulo: Ciclo 4

Instruction status:

Instruction		j	k	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	Yes	SUBD	M(A1)			Load2
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F4)	Load2	
	Mult2	No					

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
4	*FU*	Mult1	Load2		M(A1)	Add1				

- **Load2 completa; alguém esperando por Load2?**

Exemplo Tomasulo: Ciclo 5

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
2	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	No					
	Add3	No					
10	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
5	*FU*	Mult1	M(A2)		M(A1)	Add1	Mult2			

- **Timer inicia a contagem regressiva para Add1, Mult1**

Exemplo Tomasulo: Ciclo 6

IC-UNICAMP

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
1	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	Yes	ADDD		M(A2)	Add1	
	Add3	No					
9	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
6	*FU*	Mult1	M(A2)		Add2	Add1	Mult2			

- **Issue ADDD, dependência de nome em F6?**

Exemplo Tomasulo: Ciclo 7

Instruction status:

Instruction		j	k	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
0	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	Yes	ADDD		M(A2)	Add1	
	Add3	No					
8	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
7	*FU*	Mult1	M(A2)		Add2	Add1	Mult2			

- **Add1 (SUBD) completa; alguém esperando por add1?**

IC-UNICAMP

Exemplo Tomasulo: Ciclo 8

Instruction status:

Instruction		j	k	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
2	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
7	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
8	*FU*	Mult1	M(A2)		Add2	(M-M)	Mult2			

IC-UNICAMP

Exemplo Tomasulo: Ciclo 9

Instruction status:

Instruction		j	k	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
1	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
6	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
9	*FU*	Mult1	M(A2)		Add2	(M-M)	Mult2			

Exemplo Tomasulo: Ciclo 10

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
0	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
5	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
10	*FU*	Mult1	M(A2)		Add2	(M-M)	Mult2			

- **Add2 (ADDD) completa; alguém esperando por add2?**

Exemplo Tomasulo: Ciclo 11

Instruction status:

Instruction		j	k	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
4	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
11	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

- **Resultado de ADDD é escrito!**
- **Todas as instruções mais rápidas terminam neste ciclo!**

IC-UNICAMP

Exemplo Tomasulo: Ciclo 12

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
3	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
12	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

IC-UNICAMP

Exemplo Tomasulo: Ciclo 13

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
2	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
13	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

Exemplo Tomasulo: Ciclo 14

Instruction status:

Instruction		j	k	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
1	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
14	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

Exemplo Tomasulo: Ciclo 15

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
0	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
15	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

- Mult1 (MULTD) completa; alguém esperando por mult1?

Exemplo Tomasulo: Ciclo 16

IC-UNICAMP

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
40	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
16	FU	M*F4	M(A2)		(M-M+M	(M-M)	Mult2			

- **Agora é só esperar que Mult2 (DIVD) complete**

Pulando alguns ciclos (façam como exercício os ciclos faltantes?)

IC-UNICAMP

Exemplo Tomasulo: Ciclo 55

Instruction status:

Instruction		j	k	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
1	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
55	*FU*	M*F4	M(A2)		(M-M+M	(M-M)	Mult2			

Exemplo Tomasulo: Ciclo 56

IC-UNICAMP

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5	56	
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
0	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
56	FU	M*F4	M(A2)		(M-M+M	(M-M)	Mult2			

- Mult2 (DIVD) completa; alguém esperando por mult2?

IC-UNICAMP

Exemplo Tomasulo: Ciclo 57

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5	56	57
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
57	*FU*	M*F4	M(A2)		(M-M+M)	(M-M)	Result			

- **In-order issue, out-of-order execution e out-of-order completion.**

Tomasulo's Alg simulators

- http://www.dcs.ed.ac.uk/home/hase/webhase/demo/tomasulo.html
- http://www.ecs.umass.edu/ece/koren/architecture/Tomasulo/AppletTomasulo.html
  - já está com o exemplo do CAQA5 carregado, configurável

Controle do Alg. de Tomasulo

IC-UNICAMP

Fig 3.9

Instruction state	Wait until	Action or bookkeeping

Issue
FP operation
	Station r empty	`if (RegisterStat[rs].Qi¦0)`
		`{RS[r].Qj ← RegisterStat[rs].Qi}`
		`else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0};`
		`if (RegisterStat[rt].Qi¦0)`
		`{RS[r].Qk ← RegisterStat[rt].Qi`
		`else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};`
		`RS[r].Busy ← yes; RegisterStat[rd].Q ← r;`

Load or store	Buffer r empty	`if (RegisterStat[rs].Qi¦0)`
		`{RS[r].Qj ← RegisterStat[rs].Qi}`
		`else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0};`
		`RS[r].A ← imm; RS[r].Busy ← yes;`

Load only		`RegisterStat[rt].Qi ← r;`

Store only		`if (RegisterStat[rt].Qi¦0)`
		`{RS[r].Qk ← RegisterStat[rs].Qi}`
		`else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};`

Execute FP operation	`(RS[r].Qj = 0)` and `(RS[r].Qk = 0)`	Compute result: operands are in Vj and Vk
Load/store step 1	RS[r].Qj = 0 & r is head of load-store queue	`RS[r].A ← RS[r].Vj + RS[r].A;`
Load step 2	Load step 1 complete	Read from `Mem[RS[r].A]`

Write result
FP operation
or load
	Execution complete at r & CDB available	`∀x(if (RegisterStat[x].Qi=r) {Regs[x] ← result;`
		`RegisterStat[x].Qi ← 0});`
		`∀x(if (RS[x].Qj=r) {RS[x].Vj ← result;RS[x].Qj ←`
		`0});`
		`∀x(if (RS[x].Qk=r) {RS[x].Vk ← result;RS[x].Qk ←`
		`0});`
		`RS[r].Busy ← no;`

Store	Execution complete at r & `RS[r].Qk = 0`	`Mem[RS[r].A] ← RS[r].Vk;` `RS[r].Busy ← no;`

IC-UNICAMP

Exemplo: renaming em um loop

```
Loop:   L.D      F0,0(R1)
        MUL.D    F4,F0,F2
        S.D      F4,0(R1)
        DADDIU   R1,R1,-8
        BNE      R1,R2,Loop; branches if R1¦R2
```

Elementos de um vetor multiplicados por um escalar em F2

- Se “predict taken”
  - RS permitirão múltiplas execuções do loop
- Loads/Stores: podem ser executados fora de ordem desde que endereços (memória) sejam diferentes
- Se mesmo endereço:
  - se LD antes de ST, inversão causa WAR
  - se ST antes de LD, inversão causa RAW
  - inversão de ST causa WAW

IC-UNICAMP

Exemplo Loop p181

		Instruction status		
Instruction	**From iteration**	**Issue**	**Execute**	**Write result**
L.D F0,0(R1)	1	√	√	
MUL.D F4,F0,F2	1	√		
S.D F4,0(R1)	1	√		
L.D F0,0(R1)	2	√	√	
MUL.D F4,F0,F2	2	√		
S.D F4,0(R1)	2	√		

			Reservation stations				
Name	**Busy**	**Op**	**Vj**	**Vk**	**Qj**	**Qk**	**A**
Load1	Yes	Load					Regs[R1] + 0
Load2	Yes	Load					Regs[R1] – 8
Add1	No						
Add2	No						
Add3	No						
Mult1	Yes	MUL		Regs[F2]	Load1		
Mult2	Yes	MUL		Regs[F2]	Load2		
Store1	Yes	Store	Regs[R1]			Mult1	
Store2	Yes	Store	Regs[R1] – 8			Mult2	

				Register status					
Field	**F0**	**F2**	**F4**	**F6**	**F8**	**F10**	**F12**	**...**	**F30**
Qi	Load2		Mult2						

Figure 3.10 Two active iterations of the loop with no instruction yet completed. Entries in the multiplier reservation stations indicate that the outstanding loads are the sources. The store reservation stations indicate that the multiply destination is the source of the value to store.

Tomasulo: conclusão

- Algoritmo caiu no esquecimento desde o IBM/360
- Motivações para retorno
  - O algoritmo foi desenvolvido antes das caches. Mas caches hoje tem latências imprevisíveis → oportunidade para execução fora de ordem (esconder penalidades)
  - Processadores atuais mais agressivos: dificuldade de static scheduling, necessidade de dynamic scheduling, register renaming, especulação
  - O algoritmo permite grande desempenho sem exigir que o compilador produza código personalizado para uma determinada estrutura de pipeline: valuable property in na era of shrink-wrapped mass market software

3.6 Hardware-Based Speculation

- Extensão de dynamic scheduling para incorporar especulação controlada por hardware
  - 1: dynamic branch prediction
  - 2: especulação permite execução antes da resolução da control dependency (CD); com capacidade para desfazer
  - 3: dynamic scheduling → capacidade para tratar diferentes basic blocks (antes, sem especulação, somente overlap limitado entre BB)
- Seguir Data Flow Execution: executar no caminho previsto assim que operandos disponíveis

Hardware-Based Speculation (cont)

- Para especulação, necessário separar
  - bypass de valores entre instruções (especulação, não sabemos se a instrução will commit) Execute instructions along predicted execution paths but only commit the results if prediction was correct
  - commit final da instrução (mudança final de estado da máquina). Instruction commit: allowing an instruction to update the register file when instruction is no longer speculative
- Execute (out-of-order), Commit (in-order)
- Instruction Commit ≠ Instruction Complete
- Need an additional piece of hardware to prevent any irrevocable action until an instruction commits
  - I.e. updating state or taking an execution
- Reorder buffer

IC-UNICAMP

Reorder Buffer (ROB)

- Reorder buffer – holds the result of instruction between completion and commit
- É também uma extensão do Register File (RF) → pode fornecer operandos antes que a instrução commits

- Four fields:
  - Instruction type: (branch) / (store) / (register)
  - Destination field: (null) / (mem address) / (register number for loads and ALU ops)
  - Value field: output value (guardado até o commit)
  - Ready field: completed execution? (valor é válido?)

IC-UNICAMP

Reorder Buffer

- Register values and memory values are not written until an instruction commits
- On misprediction:
  - Speculated entries in ROB are cleared
- Exceptions:
  - Not recognized until it is ready to commit
- Stores: ainda 2 passos → 2º passo = instruction commit
- Renaming: antes RS agora ROB
  - mas ainda RS serve de buffer para instruções e operandos entre issue e execution
  - resultados entre execução e commit → ROB (tag agora aponta para o ROB)
- Modify reservation stations:
  - Operand source is now reorder buffer instead of functional unit

IC-UNICAMP

Tomasulo + ROB

Reorder buffer
From instruction unit
Reg #
Data
Instruction queue
FP registers
Load/store operations
Address unit
Floating-point operations
Operand buses
Load buffers
Operation bus
Store address
3
2
1
Reservation stations
2
1
Store data
Address
Memory unit
FP adders
FP multipliers
Load data
Common data bus (CDB)

Figure 3.11 The basic structure of a FP unit using Tomasulo's algorithm and extended to handle speculation. Comparing this to Figure 3.6 on page 173, which implemented Tomasulo's algorithm, the major change is the addition of the ROB and the elimination of the store buffer, whose function is integrated into the ROB. This mechanism can be extended to multiple issue by making the CDB wider to allow for multiple completions per clock.

4 passos de execução (Tom+ROB)

- Issue:
  - Se houver espaço no ROB e RS; envia operandos p RS; envia p RS nº linha do ROB
- Execute: como antes; monitora CDB e operandos
- Write result:
  - Broadcast no CDB, junto com o nº da linha do ROB; libera RS;
- Commit: (completion / graduation); depende da inst.
  - se normal commit: instrução chega ao top do ROB e valor presente → atualiza registrador e libera ROB
  - se store: idem, mas atualiza memória
  - branch: se correto → finish; se incorreto → flush ROB e re-inicia a execução da instrução correta

Exmpl p187: uso do ROB

Example Assume the same latencies for the floating-point functional units as in earlier examples: add is 2 clock cycles, multiply is 6 clock cycles, and divide is 12 clock cycles. Using the code segment below, the same one we used to generate Figure 3.8, show what the status tables look like when the MUL.D is ready to go to commit.

```
L.D      F6,32(R2)
L.D      F2,44(R3)
MUL.D    F0,F2,F4
SUB.D    F8,F2,F6
DIV.D    F10,F0,F6
ADD.D    F6,F8,F2
```

Answer Figure 3.12 shows the result in the three tables. Notice that although the SUB.D instruction has completed execution, it does not commit until the MUL.D commits. The reservation stations and register status field contain the same basic information that they did for Tomasulo's algorithm (see page 176 for a description of those fields). The differences are that reservation station numbers are replaced with ROB entry numbers in the Qj and Qk fields, as well as in the register status fields, and we have added the Dest field to the reservation stations. The Dest field designates the ROB entry that is the destination for the result produced by this reservation station entry.

IC-UNICAMP

Exmpl p187: uso do ROB (cont)

Vantagem s/ Tomasulo:

ver texto, se MUL.D gera exceção

Reorder buffer

Entry	Busy	Instruction		State	Destination	Value
1	No	L.D	F6,32(R2)	Commit	F6	Mem[32 + Regs[R2]]
2	No	L.D	F2,44(R3)	Commit	F2	Mem[44 + Regs[R3]]
3	Yes	MUL.D	F0,F2,F4	Write result	F0	#2 × Regs[F4]
4	Yes	SUB.D	F8,F2,F6	Write result	F8	#2 – #1
5	Yes	DIV.D	F10,F0,F6	Execute	F10	
6	Yes	ADD.D	F6,F8,F2	Write result	F6	#4 + #2

Reservation stations

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Load1	No							
Load2	No							
Add1	No							
Add2	No							
Add3	No							
Mult1	No	MUL.D	Mem[44 + Regs[R3]]	Regs[F4]			#3	
Mult2	Yes	DIV.D		Mem[32 + Regs[R2]]	#3		#5	

FP register status

Field	F0	F1	F2	F3	F4	F5	F6	F7	F8	F10
Reorder #	3						6		4	5
Busy	Yes	No	No	No	No	No	Yes	. . .	Yes	Yes

Figure 3.12 At the time the MUL.D is ready to commit, only the two L.D instructions have committed, although several others have completed execution. The MUL.D is at the head of the ROB, and the two L.D instructions are there only to ease understanding. The SUB.D and ADD.D instructions will not commit until the MUL.D instruction commits, although the results of the instructions are available and can be used as sources for other instructions. The DIV.D is in execution, but has not completed solely due to its longer latency than MUL.D. The Value column indicates the value being held; the format #X is used to refer to a value field of ROB entry X. Reorder buffers 1 and 2 are actually completed but are shown for informational purposes. We do not show the entries for the load/store queue, but these entries are kept in order.

IC-UNICAMP

Mesmo exemplo, sem especulação → comparar

Mostrado em slides anteriores com animação

		Instruction status		
Instruction		Issue	Execute	Write result
L.D	F6,32(R2)	√	√	√
L.D	F2,44(R3)	√	√	√
MUL.D	F0,F2,F4	√	√	
SUB.D	F8,F2,F6	√	√	√
DIV.D	F10,F0,F6	√		
ADD.D	F6,F8,F2	√	√	√

Reservation stations

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	No						
Load2	No						
Add1	No						
Add2	No						
Add3	No						
Mult1	Yes	MUL	Mem[44 + Regs[R3]]	Regs[F4]			
Mult2	Yes	DIV		Mem[32 + Regs[R2]]	Mult1		

Register status

Field	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi	Mult1					Mult2			

Figure 3.8 Multiply and divide are the only instructions not finished.

IC-UNICAMP

Exmpl p189: ROB e loop

Example Consider the code example used earlier for Tomasulo's algorithm and shown in Figure 3.10 in execution:

```
Loop:   L.D      F0,0(R1)
        MUL.D    F4,F0,F2
        S.D      F4,0(R1)
        DADDIU   R1,R1,#-8
        BNE      R1,R2,Loop    ;branches if R1¦R2
```

Assume that we have issued all the instructions in the loop twice. Let's also assume that the L.D and MUL.D from the first iteration have committed and all other instructions have completed execution. Normally, the store would wait in the ROB for both the effective address operand (R1 in this example) and the value (F4 in this example). Since we are only considering the floating-point pipeline, assume the effective address for the store is computed by the time the instruction is issued.

Answer Figure 3.13 shows the result in two tables.

Exmpl p189: ROB e loop (cont

Reorder buffer

Entry	Busy	Instruction	State	Destination	Value
1	No	L.D F0,0(R1)	Commit	F0	Mem[0 + Regs[R1]]
2	No	MUL.D F4,F0,F2	Commit	F4	#1 × Regs[F2]
3	Yes	S.D F4,0(R1)	Write result	0 + Regs[R1]	#2
4	Yes	DADDIU R1,R1,#-8	Write result	R1	Regs[R1] – 8
5	Yes	BNE R1,R2,Loop	Write result		
6	Yes	L.D F0,0(R1)	Write result	F0	Mem[#4]
7	Yes	MUL.D F4,F0,F2	Write result	F4	#6 × Regs[F2]
8	Yes	S.D F4,0(R1)	Write result	0 + #4	#7
9	Yes	DADDIU R1,R1,#-8	Write result	R1	#4 – 8
10	Yes	BNE R1,R2,Loop	Write result		

FP register status

Field	F0	F1	F2	F3	F4	F5	F6	F7	F8
Reorder #	6				7				
Busy	Yes	No	No	No	Yes	No	No	...	No

Figure 3.13 Only the L.D and MUL.D instructions have committed, although all the others have completed execution. Hence, no reservation stations are busy and none is shown. The remaining instructions will be committed as quickly as possible. The first two reorder buffers are empty, but are shown for completeness.

Exmpl p189: ROB e loop (cont)

- Supor BNE not taken na 1ª iteração
  - instruções anteriores commit quando chegarem ao top do ROB; quando BNE chegar ao top → clear somente
- Se predição errada
  - clear ROB: todas as instruções depois do branch
  - permite que as instruções anteriores ao branch commit
  - re-inicia o fetch do endereço correto
- Exceções
  - não reconhecidas até o momento do commit
  - registro da exceção fica no ROB até o commit
    - se instrução causadora está em ramo errado de branch, simplesmente flush
    - se ela chega do top do ROB, então commit e trata exceção

Hazards e o ROB

- WAW e WAR through memory
  - ok com especulação e commit em ordem
- RAW through memory, ok com duas restrições
  - Não permitir 2º passo de Load se existir Store no ROB com “destination field” = campo A do Load
  - Cálculo de endereços efetivo de Loads feito em ordem com relação a todos os Stores anteriores
  - (qualquer Load que acesse posição de memória escrita por Store anterior até que o Store tenha escrito o valor)

IC-UNICAMP

3.7 Multiple Issue and Static Scheduling

- Técnicas vistas: eliminar hazards e fazer CPI → 1
- To achieve CPI < 1, multiple instructions per clock
- Solutions:
  - Com superescalar: número variado de instruções por clock
    - Statically scheduled superscalar processors: in-order execution (statically scheduled)
    - Dynamically scheduled superscalar processors: out-of-order execution (dynamically scheduled)
  - VLIW (very long instruction word) processors
    - número fixo de instruções empacotadas, contendo indicações explícitas de paralelismo entre instruções
    - escalonamento inerentemente estático pelo compilador (ex EPIC: ExplicitlyParallel Instruction Computer)
- Statically scheduled superscalar e VLIW: compilador
- Uso comum
  - Narrow issue: Statically scheduled superscalar
  - Wide issue: VLIW ou Dynamically scheduled superscalar

Multiple Issue Approaches

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the ARM Coretex A8
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

Figure 3.15 The five primary approaches in use for multiple-issue processors and the primary characteristics that distinguish them. This chapter has focused on the hardware-intensive techniques, which are all some form of superscalar. Appendix H focuses on compiler-based approaches. The EPIC approach, as embodied in the IA-64 architecture, extends many of the concepts of the early VLIW approaches, providing a blend of static and dynamic approaches.

VLIW Processors

- Package multiple operations into one instruction
- Example VLIW processor:
  - One integer instruction (or branch)
  - Two independent floating-point operations
  - Two independent memory references
- Instruções: conjunto de campos por FU (16-24 bits). Instrução total 80-120 bits.
  - Itanium tem 6 operações por pacote
- Must be enough parallelism in code to fill the available slots
  - loop unrolling e outras técnicas

Exmpl p195: loop unrolling in VLIW

Example Suppose we have a VLIW that could issue two memory references, two FP operations, and one integer operation or branch in every clock cycle. Show an unrolled version of the loop x[i] = x[i] + s (see page 158 for the MIPS code) for such a processor. Unroll as many times as necessary to eliminate any stalls. Ignore delayed branches.

Answer Figure 3.16 shows the code. The loop has been unrolled to make seven copies of the body, which eliminates all stalls (i.e., completely empty issue cycles), and runs in 9 cycles. This code yields a running rate of seven results in 9 cycles, or 1.29 cycles per result, nearly twice as fast as the two-issue superscalar of Section 3.2 that used unrolled and scheduled code.

IC-UNICAMP

Exmpl p195: loop unrolling in VLIW (2)

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
L.D F0,0(R1)	L.D F6,-8(R1)			
L.D F10,-16(R1)	L.D F14,-24(R1)			
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2	
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2	
		ADD.D F20,F18,F2	ADD.D F24,F22,F2	
S.D F4,0(R1)	S.D F8,-8(R1)	ADD.D F28,F26,F2		
S.D F12,-16(R1)	S.D F16,-24(R1)			DADDUI R1,R1,#-56
S.D F20,24(R1)	S.D F24,16(R1)			
S.D F28,8(R1)				BNE R1,R2,Loop

Figure 3.16 VLIW instructions that occupy the inner loop and replace the unrolled sequence. This code takes 9 cycles assuming no branch delay; normally the branch delay would also need to be scheduled. The issue rate is 23 operations in 9 clock cycles, or 2.5 operations per cycle. The efficiency, the percentage of available slots that contained an operation, is about 60%. To achieve this issue rate requires a larger number of registers than MIPS would normally use in this loop. The VLIW code sequence above requires at least eight FP registers, while the same code sequence for the base MIPS processor can use as few as two FP registers or as many as five when unrolled and scheduled.

VLIW Processors

- Disadvantages:
  - Statically finding parallelism
  - Code size: principalmente devido a loop unrolling
  - No hazard detection hardware (early VLIW)
    - blocking caches → stalls
    - VLIW mais recentes: hw for hazard detection
  - Binary code compatibility
    - different number of FU and latencies require different version of code