Arquitetura e Organização de Computadores II
Aula 2

ROTEIRO

2.3 Escalonamento dinâmico de instruções
2.3.1 Algoritmo de Tomasulo
2.3.2 Renomeação de registradores

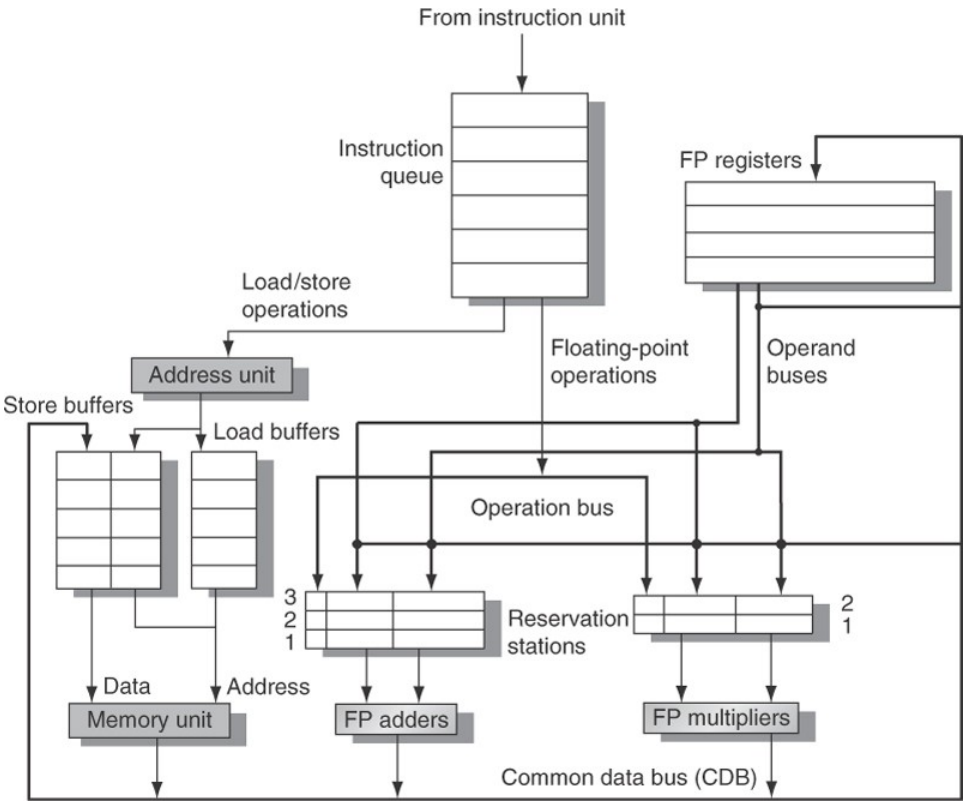
NOTAS DE AULA

FC

Computer Organization: A Quantitative Approach
John L. Hennessy
David A. Patterson
5 Edition

Páginas 148 - 156

I. Arquitetura Básica utilizando o Algoritmo de Tomasulo



** Instruções são enviadas para a fila de instrução, da qual elas são emitidas utilizando a política FIFO. As estações de reserva armazena a operação, os operandos e informações necessárias para detectar possíveis limitações (conflitos). O buffer de load possui três funcionalidades: (1) manter os componentes do endereço até ele ser calculado; (2) manter a gerencia dos loads que estão

aguardando para serem processados; e (3) manter o resultados dos loads que estam aguardando pelo CDB. Os stores são gerenciados similarmente aos loads. Os resultados das unidades FP e load são colocados no CDB, que alimenta as estações de reservas e o buffer de reordenação. Os somadores implementam soma e subtração e os multiplicadores implementam multiplicação e divisão.

** Fases da execução

1. Issue: Emite uma instrução se existe uma estação de reserva e um slot no ROB livres.
2. Execute: Processa a instrução (unidade funcional).
3. Write result: Escreve o resultado no CDB, e do CDB nas estações de reserva, registradores e buffer de store.

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;`

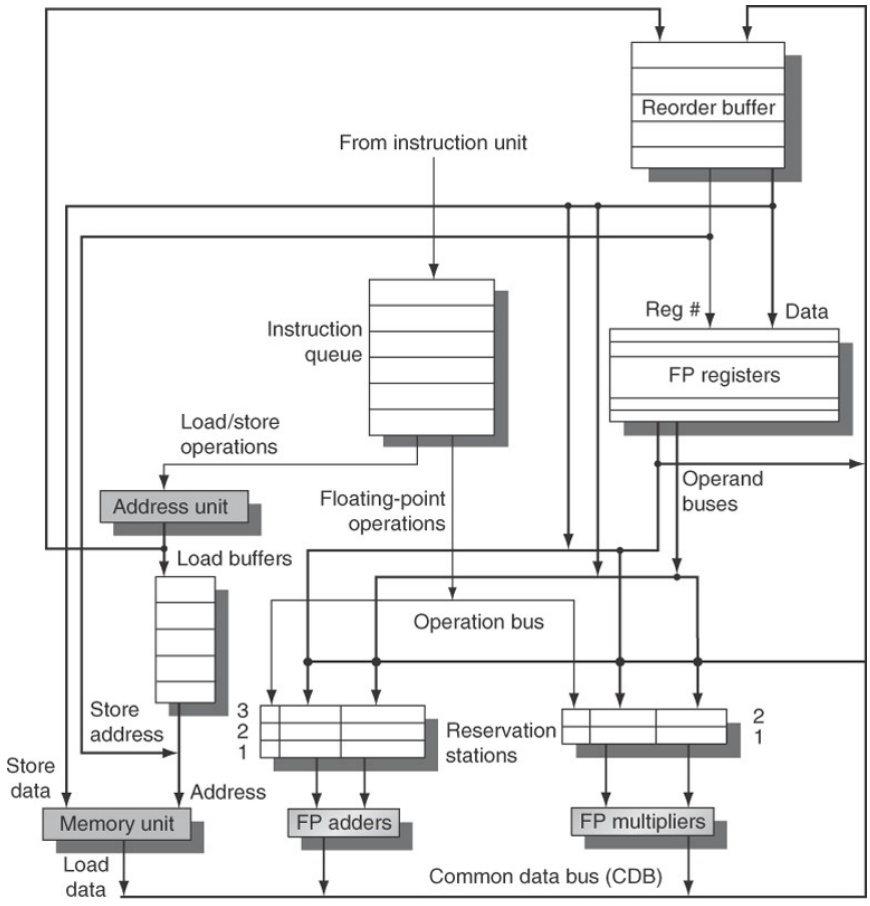
Figure 3.9 Steps in the algorithm and what is required for each step. For the issuing instruction, rd is the destination, rs and rt are the source register numbers, imm is the sign-extended immediate field, and r is the reservation station or buffer that the instruction is assigned to. RS is the reservation station data structure. The value returned by an FP unit or by the load unit is called result. RegisterStat is the register status data structure (not the register file, which is Regs[]). When an instruction is issued, the destination register has its Qi field set to the number of the buffer or reservation station to which the instruction is issued. If the operands are available in the registers, they are stored in the V fields. Otherwise, the Q fields are set to indicate the reservation station that will produce the values needed as source operands. The instruction waits at the reservation station until both its operands are available, indicated by zero in the Q fields. The Q fields are set to zero either when this instruction is issued or when an instruction on which this instruction depends completes and does its write back. When an instruction has finished execution and the CDB is available, it can do its write back. All the buffers, registers, and reservation stations whose values of Qj or Qk are the same as the completing reservation station update their values from the CDB and mark the Q fields to indicate that values have been received. Thus, the CDB can broadcast its result to many destinations in a single clock cycle, and if the waiting instructions have their operands they can all begin execution on the next clock cycle. Loads go through two steps in execute, and stores perform slightly differently during write result, where they may have to wait for the value to store. Remember that, to preserve exception behavior, instructions should not be allowed to execute if a branch that is earlier in program order has not yet completed. Because any concept of program order is not maintained after the issue stage, this restriction is usually implemented by preventing any instruction from leaving the issue step, if there is a pending branch already in the pipeline. In Section 3.6, we will see how speculation support removes this restriction.

II. As Estações de Reserva e o Banco de Registradores

** Cada estação de reverva tem sete campos
1. Op: operação
2. Qj e Qk: estações de reserva que produzirão os operandos (valor ZERO indica que os operandos estão disponíveis em Vj e Vk.
3. Vj, e Vk: valores dos operandos.
4. A: informação para cálculo de endereço (load). Inicialmente, armazena o imediato da instrução; após o cálculo do endereço armazena o endereço efetivo.
5. Busy: para indicar que a estação está ocupada.

** Cada registrador do banco tem um campo Qi, que indica qual estação de reserva contém a operação cujo resultado deve ser armazenado neste registrador.

III. Arquitetura Básica utilizando o Algoritmo de Tomasulo com Especulação



*** Fases da execução

1. Issue: Emite uma instrução se existe uma estação de reserva e um slot no ROB livres.
2. Execute: Processa a instrução (unidade funcional).
3. Write result: Escreve o resultado no CDB, e do CDB no ROB.
4. Commit: Escreve o resultado no destino final (registrador ou memória).

Status	Wait until	Action or bookkeeping
Issue all instructions	Reservation station (r) and ROB (b) both available	`if (RegisterStat[rs].Busy)/*in-flight instr. writes rs*/ {h ← RegisterStat[rs].Reorder; if (ROB[h].Ready)/* Instr completed already */ {RS[r].Vj ← ROB[h].Value; RS[r].Qj ← 0;} else {RS[r].Qj ← h;} /* wait for instruction */ } else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0;}; RS[r].Busy ← yes; RS[r].Dest ← b; ROB[b].Instruction ← opcode; ROB[b].Dest ← rd;ROB[b].Ready ← no;`
FP operations and stores		`if (RegisterStat[rt].Busy) /*in-flight instr writes rt*/ {h ← RegisterStat[rt].Reorder; if (ROB[h].Ready)/* Instr completed already */ {RS[r].Vk ← ROB[h].Value; RS[r].Qk ← 0;} else {RS[r].Qk ← h;} /* wait for instruction */ } else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0;};`
FP operations		`RegisterStat[rd].Reorder ← b; RegisterStat[rd].Busy ← yes; ROB[b].Dest ← rd;`
Loads		`RS[r].A ← imm; RegisterStat[rt].Reorder ← b; RegisterStat[rt].Busy ← yes; ROB[b].Dest ← rt;`
Stores		`RS[r].A ← imm;`
Execute FP op	(RS[r].Qj == 0) and (RS[r].Qk == 0)	Compute results—operands are in Vj and Vk
Load step 1	(RS[r].Qj == 0) and there are no stores earlier in the queue	`RS[r].A ← RS[r].Vj + RS[r].A;`
Load step 2	Load step 1 done and all stores earlier in ROB have different address	`Read from Mem[RS[r].A]`
Store	(RS[r].Qj == 0) and store at queue head	`ROB[h].Address ← RS[r].Vj + RS[r].A;`
Write result all but store	Execution done at *r* and CDB available	`b ← RS[r].Dest; RS[r].Busy ← no; ∀x(if (RS[x].Qj==b) {RS[x].Vj ← result; RS[x].Qj ← 0}); ∀x(if (RS[x].Qk==b) {RS[x].Vk ← result; RS[x].Qk ← 0}); ROB[b].Value ← result; ROB[b].Ready ← yes;`
Store	Execution done at *r* and (RS[r].Qk == 0)	`ROB[h].Value ← RS[r].Vk;`
Commit	Instruction is at the head of the ROB (entry h) and ROB[h].ready == yes	`d ← ROB[h].Dest; /* register dest, if exists */ if (ROB[h].Instruction==Branch) {if (branch is mispredicted) {clear ROB[h], RegisterStat; fetch branch dest;};} else if (ROB[h].Instruction==Store) {Mem[ROB[h].Destination] ← ROB[h].Value;} else /* put the result in the register destination */ {Regs[d] ← ROB[h].Value;}; ROB[h].Busy ← no; /* free up ROB entry */ /* free up dest register if no one else writing it */ if (RegisterStat[d].Reorder==h) {RegisterStat[d].Busy ← no;};`

Figure 3.14 Steps in the algorithm and what is required for each step. For the issuing instruction, rd is the destination, rs and rt are the sources, r is the reservation station allocated, b is the assigned ROB entry, and h is the head entry of the ROB. RS is the reservation station data structure. The value returned by a reservation station is called the result. RegisterStat is the register data structure, Regs represents the actual registers, and ROB is the reorder buffer data structure.

IV. Prática

Após implementar a hierarquia de memória, a próxima etapa é implementar um clock, um dispositivo de entrada e saída e o processador.

- Clock
    - Este componente é responsável por enviar pulsos aos componetes do sistema.

- Dispositivo de entrada e saída
    - Este componente simula um terminal para entrada e saída de dados.

- Processador

- Este componente dever ser um simulador de uma arquitetura básica que utiliza o algoritmo de Tomasulo SEM ESPECULAÇÃO.

- Sua implementação deve eguir a especificação contida em:

Computer Architecture: A Quantitative Approach
Hennessy e Patterson
5 Edition
Seção 3.5

MM
Implemente e teste os novos componentes do simulador.