Microprocessor architecture

- Instruction set architecture
- CISC architecture
- RISC architecture
- Pipelining
- Superscalar and superpipelined architectures
- Post-RISC architecture
- Out-of-order execution
- VLIW architecture

1

Instruction set architecture

- The part of the processor that is visible to the (assembly language) programmer or compiler writer
  - defines the instructions, registers and mechanisms to access memory that the processor can use to operate on data
- Specifies the
  - registers
  - machine instructions
  - memory addresses
  - addressing modes
- Example: Intel IA-32
  - defines a family of microprocessors, starting from 8086 (1978) to the Pentium 4 (2000)
  - all binary compatible (within certain limits)

2

Registers

- Registers are memory locations in which the processor stores data that it operates on
  - implemented by very fast memory technology
- Most modern microprocessors use a number of general purpose registers
- *Register-memory architecture*
  - operations can access both registers and memory
- *Load-store architecture*
  - operations can only be performed on registers
  - memory can only be accessed with load or store operations
- Number of registers vary
  - from about 10 to over 200

3

Machine instructions

- The instruction set specifies the machine instructions that the processor can execute
  - expressed as assembly language instructions
- Instructions can have 2 or 3 operands
  - `add a,b` $a \leftarrow a + b$, result overwrites a
  - `add c,a,b` $c \leftarrow a + b$, result placed in c
- Nr of memory references in an instruction can be
  - 0 – load/store (RISC)
  - 1 – Intel 80x86, Motorola 68000
  - 2 or 3, CISC architectures
- Translated to binary machine code (opcodes) by an assembler
  - machine instructions can be of different lengths

C code

```
c = a + b
```

Assembly language register – memory

```
load  R1, a
add   R1, b
store c, R1
```

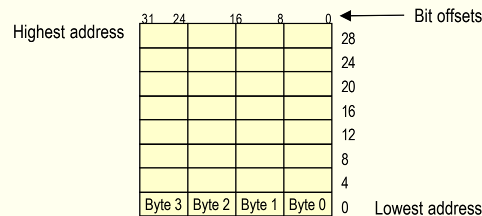
Assembly language load – store

```
load  R1, a
load  R2, b
add   R1, R2
store c, R1
```

4

Memory addressing

- Contigous, byte-addressable memory
- Can address
  - byte (8 bits)
  - halfword (16 bits)
  - word (32 bits)
  - double word (64 bits)

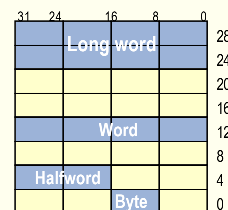


- Little endian
  - bytes of a word are numbered starting from the least significant byte
  - IA-32 architecture is little endian
- Big endian
  - bytes of a word are numbered starting from the most significant byte

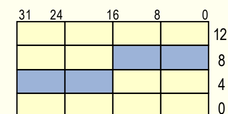
5

Memory alignment

- An object of size *S* bytes at (byte) address *A* is memory aligned if *A mod S = 0*
  - bytes are always aligned
  - halfwords are aligned at even byte addresses
  - words are aligned at byte offsets 0 and 4
  - double words are aligned at byte offsets 0



- Misaligned data
  - *Example*: a word located at byte offset 6
- Misaligned data can cause performance degradation
  - most compilers can automatically align data



6

Addressing modes

- The addressing modes describe how the processor can specify the address of an object
  - can specify constant values, registers or memory locations
- Immediate
  - `add R1, #4` [R1] ← [R1]+4
- Register
  - `add R1, R2` [R1] ← [R1]+[R2]
- Displacement
  - `add R1, 20(R2)` [R1] ← [R1]+Mem(20+[R2])
- Indirect
  - `add R1,(R2)` [R1] ← [R1]+Mem([R2])
- Indexed
  - `add R1,(R1+R2)` [R1] ← [R1]+Mem([R1]+[R2])

7

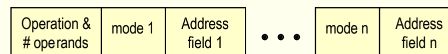
Addressing modes (cont.)

- Direct or absolute
  - `add R1,(2124)` [R1] ← [R1]+Mem(2124)
- Memory indirect
  - `add R1, @(R2)` [R1] ← [R1]+Mem(Mem([R2]))
- Autoincrement
  - `add R1, (R2)+` [R1] ← [R1]+Mem([R2])
[R2] ← [R2]+*d*
- Autodecrement
  - `add R1, -(R2)` [R2] ← [R2]-*d*
[R1] ← [R1]+Mem([R2])
- Scaled
  - `add R1,100(R2)[R3]` [R1] ← [R1]+ Mem(100+[R2]+[R3]**d*)

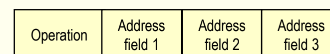
8

Instruction encoding

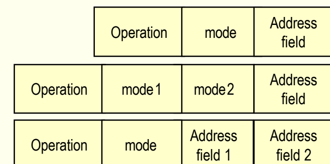
- Assembly language instructios are encoded into numerical machine instructions by the assembler
- Instruction formats can be of three different types
  - variable, fixed or hybrid
- Variable length
  - supports any number of operands



- Fixed format
  - always the same number of operands
  - addressing mode specified as part of opcode



- Hybrid format
  - multiple formats, depending on the operation



CISC architecture

- Complex Instruction Set Computer
  - large instruction set
  - instructions can perform very complex operations
  - variable instruction formats: 16, 32 or 64 bits
  - large number of addressing modes
  - few registers
- Powerful assembly language
  - designed so that high-level language constructs could be compiled into as few assembly instructions as possible
- Implemented using microcode
- Example: Motorola MC68000 family
  - 18 different address modes in a MOVE-instruction

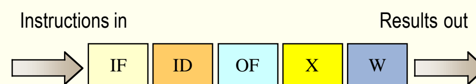
RISC architecture

- Reduced Instruction Set Computer
- Characteristic for a RISC processor is
  - no microcode
  - relatively few instructions
  - simple addressing modes
  - only load/store instructions access memory
  - uniform instruction length
  - more registers than CISC processors
  - pipelined instruction execution
  - delayed branching
- Examples: SPARC, MIPS, HP-PA, Alpha, PowerPC

11

Instruction pipelining

- Instruction execution is divided into a number of stages
  - instruction fetch
  - instruction decode
  - operand fetch
  - execute
  - writeback

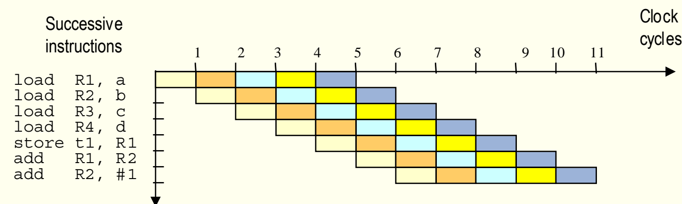


- The time to move an instruction one step through the pipeline is called a machine cycle
  - can complete one instruction everey cycle
  - without pipelining we could complete one instruction every 5 cycles
- CPI – Clock cycles Per Instruction
  - the number of cycles needed to execute an instruction
  - different for different instructions

12

Pipelined instruction execution

- All pipeline stages can execute in parallel
  - separate hardware units for each stage



- After 5 clock cycles, the pipeline is full
  - finishes one instruction every clock period
  - it takes 5 clock periods to finish one instruction
- Pipelining increases the CPU instruction throughput
  - does not reduce the time to execute one instruction

13

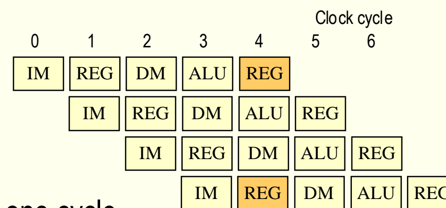
Pipeline hazards

- Situations that prevent the next instruction in the stream from executing during its clock cycle
- Structural hazards
  - arise from resource conflicts
  - two instructions need the same functional unit in the same pipeline stage
- Data hazards
  - arise when an instruction depends on the result of a previous instruction, which has not completed yet
- Control hazards
  - arise from branches in the instruction stream
- Hazards force the pipeline to stall
  - stop the instruction fetch for a number of cycles until we have all resources needed to continue

14

Structural hazards

- Each stage of the pipeline is handled by a separate functional unit
  - instruction fetch uses the instruction memory
  - instruction decode uses the program counter register
  - operand fetch uses the data memory
  - execute uses the ALU
  - writeback uses the registers
- *Example*: two instructions need access to the registers in the same clock cycle
  - the last instruction will stall for one cycle



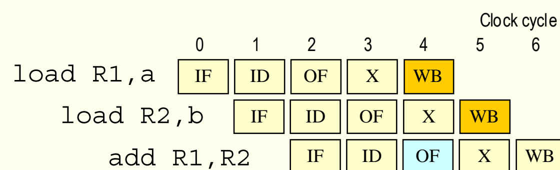
15

Data hazards

- An instruction depends on the result of a previous instruction, which has not completed yet
- Example:
  - the add-operation accesses R1 and R2 in cycle 4
  - the load-operations write the value into register R2 in the write-back stage
  - R1 ready in cycle 4
  - R2 ready in cycle 5

```
load  R1, a
load  R2, b
add   R1, R2
```

- The add must stall for one cycle

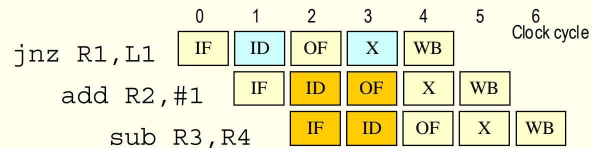


16

Control hazards

- Branch instructions transfer control in the program execution
  - may assign a new value to the PC
- Conditional branches may be taken or not taken
  - a taken branch assigns the target address to the PC
  - a branch that is not taken (falls through) continues at the next instruction
- The instruction is recognized as a branch in the instruction decode phase
- Can decide whether the branch will be taken or not in the execute stage

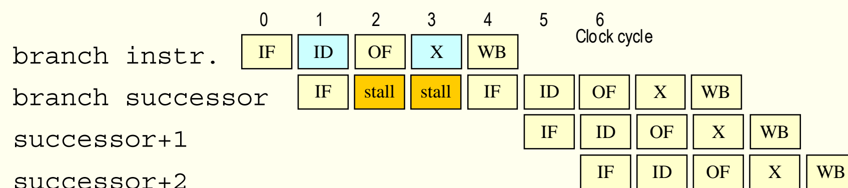
```
      jnz R1,L1
      add R2,#1
      sub R3,R4
L1:
      mov R1,#0
```



17

Stalling the pipeline

- One way of executing branch instructions is to stall the pipeline for 2 cycles when a branch instruction is decoded
  - wait until we know the outcome of the branch
- The instruction executed after the branch is either the `add` or the `mov` instruction
  - we don't know which before the branch instruction has reached execution stage in the pipeline

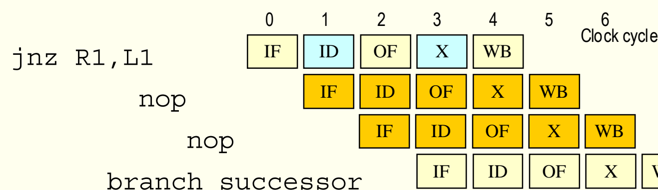


18

Branch delay slots

- To simplify branch exeution we can insert a delay after a branch instruction
  - forces the execution to wait until the outcome of the branch is known
- Insert `nop` instructions after a branch
  - called a branch delay slot

```
     jnz R1,L1
     nop
     nop
     add R2,#1
     sub R3,R4
L1:
     mov R1,#0
```



19

Delayed branches

- More efficient is to use the branch delay slot for useful work
  - instead of `nop` instructions we execute useful instructions in the branch delay slot
- These instructions are always executed regardless of how the branch goes
  - can be useful instructions or at least harmless instructions
  - `nop's` can also be used
- Possible to use the branch delay slot to compute something that will be needed
- Branch instructions are not allowed

```
     jnz R1,L1
     mov R1,a
     mov R5,b
     add R2,#1
     sub R3,R4
L1:
     add R1,R5
     sto c,R1
```

20

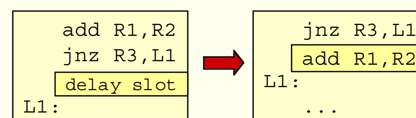
Branch prediction

- Guess the outcome of the branch
- Predict as not taken
  - we assume the branch will not be taken
  - continue the execution with the instruction following the branch
  - if the branch turns out not to be taken, then we guessed right and continue
  - the branch instruction behaves like a `nop`
- If the prediction was wrong, we have to undo the effects of the falsely executed instructions
  - not allowed to change the state of the processor until the branch outcome is known (no writeback)
  - flush out the mispredicted instructions from the pipeline
  - restart the instruction fetch from the branch target

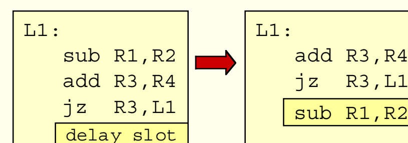
21

Scheduling branch delay slots

- Three ways of scheduling instructions into branch delay slot
- From code before the branch
  - the branch may not depend on the rescheduled instruction
  - always improves performance



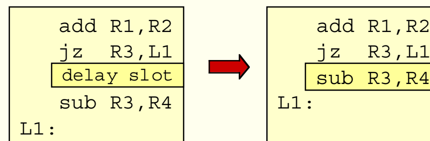
- From the target of the branch
  - must be correct to execute the instruction also if the branch is not taken
  - may need to duplicate instructions
  - improves performance when the branch is taken



22

Scheduling branch delay slots (cont.)

- From the fall through code
  - must be correct to execute the instruction also if the branch is taken
  - improves performance when the branch is not taken

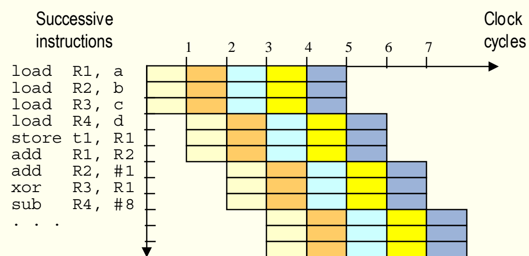


Cancelling branch

- Many architectures with branch delay slots have a cancelling (or nullifying) branch instruction
- The encoding of the branch instruction includes the direction the branch was predicted
  - taken or not taken
- When the branch behaves as predicted, the instruction in the branch delay slot is executed as a normal delayed branch
- When the branch is incorrectly predicted, the instruction in the branch delay slot is modified into a `nop` instruction
- Can use all three methods of scheduling instructions into the branch delay slot

Superscalar architecture

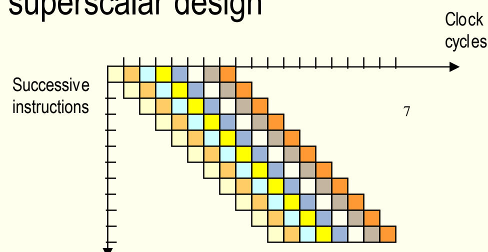
- Increases the ability of the processor to use instruction level parallelism
- Multiple instructions are issued every cycle
  - multiple pipelines operating in parallel
- Example:
  - 3 parallel pipelines each with 5 stages
  - 3-way superscalar processor



25

Superpipelined architecture

- The instruction execution pipeline is divided into a large number of simple stages
  - deep pipeline
  - higher clock frequency
- Pipeline clock cycle can be a multiple of the processor clock cycle
- Often combined with a superscalar design



26

Post-RISC architecture

- Modern processors have developed further from the basic ideas behind RISC architecture
  - exploit more instruction level parallelism
- Characteristics:
  - parallel instruction execution (superscalar)
  - deep pipeline (superpipelined)
  - extended instruction set
  - out-of-order execution
  - branch prediction
  - register renaming

27

In-order execution

- Instructions are executed in program order
  - limits the opportunities for instruction level parallelism
- Dependecies between instructions force them to be executed in program order
  - the add instruction uses the value loaded into R1
  - the store instruction uses the value produced by the add
  - the sub instruction modifies the value in R0
  - the branch condition depends on R0
- The chain of dependencies can be as long as the entire program

```
L1:
    load R1,(R0)
    add  R1,R2
    sto  (R0),R1
    sub  R0,#8
    jnz  R0,L1
```

28

Out-of-order instruction execution

- To be able to use more ILP we allow the processor to execute instruction out of order
  - also called dynamic instruction execution or dynamic instruction scheduling
- Have to guarantee that the program produces the same result as if executed in order
- To make out-of-order execution possible we have to eliminate dependences in the code
- Three types of dependences:
  - data dependences
  - name dependences
  - control dependences

29

Data dependence

- An instruction *j* depends on data from a previous instruction *i*
  - can not execute *j* before the earlier instruction *i*
  - can not execute *i* and *j* simultaneously
- Data dependences are properties of the program
  - wether this leads to a data hazard and a pipeline stall depends on the pipeline organization
- We can overcome problems of data dependences by
  - sceduling instructions so that dependences do not cause hazards
  - transforming the code to eliminate the dependance
- Loop unrolling can eliminate dependences
  - also removes branches and gives mor opportunities for instruction scheduling

30

Name dependence

- Two (or more) instructions use the same register, but there is no data transfer between the instructions
- Two types of name dependences
  - assume we have two instructions *i* and *j*, in this order
- Output dependence
  - instructions *i* and *j* write to the same register or memory location
- Antidependence
  - instruction *j* writes a register or memory location that instruction *i* reads
- The instructions can be executed in parallel if we choose other registers for the operations

```
load R0,c
add  R0,#1
sto  c,R0
load R0,d
add  R0,#1
sto  d,R0
```

```
add  (R0),R1
load R0,c
...
```

31

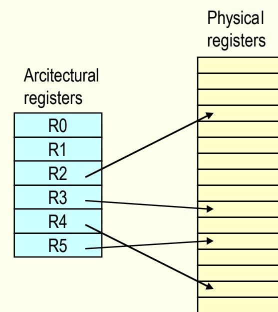
Control dependence

- Control dependences determine the ordering of an instruction with respect to a branch instruction
  - if the branch is taken, the instruction is executed
  - if the branch is not taken, the instruction is not executed
- An instruction that is control dependent on a branch can not be moved before the branch
  - instructions from the *then*-part of an *if*-statement can not be executed before the branch
- An instruction that is not control dependant on a branch can not be moved after the branch
  - other instructions can not be moved into the *then*-part of an *if*-statement
- Can lift these restrictions by using branch prediction and speculative execution

32

Register renaming

- The instruction set architecture defines a set of logical registers visible to the (assembly language) programmer
  - general-purpose registers
  - special registers (IP, SP, ...)
- The pipeline execution uses a much larger set of internal physical registers for use in program execution
  - register renaming dynamically associates physical registers to logical registers
  - removes name dependencies
- Register renaming can be done already in the instruction decode phase



33

Rotating registers

- Rotating registers help to avoid dependencies in loops
- Example: copying elements between two arrays
  - counter i in R0
  - length of arrays N in R1
  - address of X in R2
  - address of Y i R3
- Dependencies may introduce stalls
  - store can not start before the load is ready
- But the assignments could all be done in parallel
  - no dependencies between the iterations

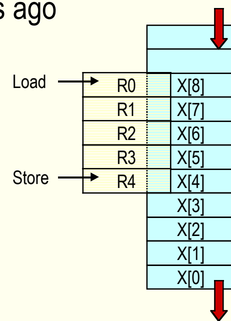
```
for (i=0;i<N;i++){
    Y[i] = X[i]
}
```

```
    mov   R0,#0
    mov   R1,N
    mul   R1,#d
L1:
    load  R4,R0(R2)
    store R0(R3),R4
    add   R0,#d
    cmp   R0,R1
    jne   L1
```

34

Rotating registers (cont.)

- The registers are renumbered for every loop iteration
  - data in R1 in the first iteration will in the second iteration be in R2
- The array copying can then be implementd like:
  (using pseudo assembler code)
- The store can start immediately
  - stores data that was loaded four iterations ago




```
load X[0] into R4
load X[1] into R3
load X[2] into R2
load X[3] into R1
for (i=0,i<N-4;i++) {
   load  R0, X[i+4]
   store Y[i], R4
}
store R3 to Y[N-4]
store R2 to Y[N-3]
store R1 to Y[N-2]
store R0 to Y[N-1]
```

35

Dynamic branch prediction

- So far we have only presented methods for static branch prediction
  - the prediction does not depend on the dynamic behaviour of the program
  - predict as taken
  - predict backwards branches as taken and forward brances as not taken
- In dynamic branch prediction we base the prediction on the outcome of the branch earlier in the execution
  - collect branch history information, on which we base the prediction
- In practice it is not possible to store information about all branches in a program
  - no upper limit on the number of branches

36

Branch history

- Branch history information is collected in a small cache memory called the *branch history table*
  - memory address of the branch instruction
  - branch history information (taken/not taken)
- In its most simple implementation, entries in the branch history table are indexed by the lower (least significant) part of the branch instruction address
  - two branches may use the same table entry
- Stores the outcome of the most recent branch executions
  - need at least one bit in each table entry to store the outcome of the branch (taken / not taken)
  - if no branch history exists, use static prediction

37

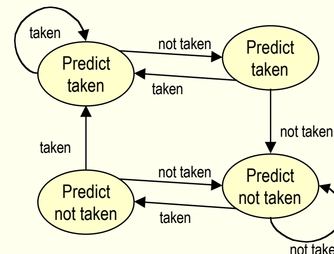
One bit branch history

- Predict that the branch goes the same way as the last time it was executed
  - if the prediction turns out to be wrong, invert the prediction bit
- Mispredicts both the first and the last iteration of a loop
  - misprediction of the last iteration is inevitable, since the branch has been taken *N-1* times (in a loop of length *N*)
  - after executing the last, mispredicted, iteration of the loop the prediction bit is set to false
  - causes a misprediction in the first iteration when we execute the loop the next time

38

N-bit branch history

- Use two bits to store branch history
  - a prediction must miss twice before it is changed
  - gives four different states

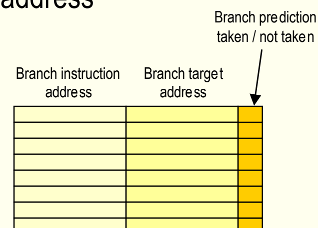


- In general, we can use *N* bits for the branch history
  - the counter takes values between 0 and 2^N-1
  - incremented if branch is taken, decremented if branch is not taken
  - if the counter is greater or equal than half of the maximum value, we predict the branch as taken, otherwise not taken
- In practice, 2 bits is enough for accurate branch prediction

39

Branch target buffer

- The branch target buffer stores also the target address of the branch
  - we can find the target address of the branch already in the instruction fetch phase
  - if the branch is predicted as taken, we can immediately start fetching instructions from the branch target address
- Implemented by associative memory
- One alternative is to only store information about branches that are predicted to be taken
  - if we find an entry in the table, it is predicted as taken
  - otherwise, we predict it as not taken and continue execution with the next instruction
  - used with one-bit branch history



40

Predicting call/returns

- Procedure calls are unconditional branches
  - always taken
- Procedure calls and returns are paired
  - one return for each procedure call
  - can have nested procedure calls
- Can use a return address stack (RAS) as a branch target buffer to predict the return address
  - push the return address when the call instruction is executed
  - pop it when the return instruction is executed

Dynamic scheduling

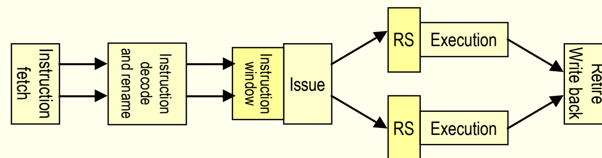
- In dynamic scheduling instructions are rearranged so that the pipelines are kept busy
- The pipeline is allowed to rearrange the instructions to avoid hazards
  - makes it easier for the compiler to produce well optimized code
  - allows code optimized for one processor to execute efficiently on another processor
- Example:
  - the add depends on the result of the division
  - the load-instruction stalls until the div and add are ready

```
div  R0,R1
add  R2,R0
load R5,a
sub  R5,R6
```

- No dependencies between div/add and load/sub
  - can execute the load/sub befor the div/add

Out-of-order execution

- The instruction fetch, execution and retirement is separated from each other
  - instructions are fetched and decoded in order
  - instructions are executed out of order
  - results of the execution are retired in order
- Instructions can be executed when all operands are available and a functional unit for the operation is available
  - result of execution is stored in internal registers
  - retired in program order, written back to registers or memory



43

Tomasulo's algorithm

- Method for dynamic instruction scheduling
  - out-of-order instruction execution
  - R.M. Tomasulo, An Efficient Algorithm for Exploiting Multiple Arithmetic Units, *IBM J. of Res.&Dev. 11:1* (Jan 1967)
  - developed for IBM 360/91
- Similar out-of-order execution used in Alpha 21264, HP 8000, MIPS R10000, Pentium II and PowerPC 604
- Aviods pipeline stalls due to dependencies
  - instructions whose operands are available can execute out of order
- Combines
  - register renaming
  - out-of-order instruction execution
  - data forwarding (short circuiting)

44

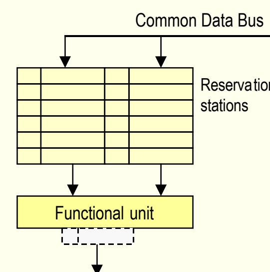
Reservation stations

- Buffer area for each functional unit
  - holds instructions to be executed
  - each functional unit has its own set of reservation stations
- Contains
  - instructions that have been issued and which are waiting to be executed by the functional unit
  - operands of the instruction, if these are available
  - the source of the operands if they are not yet available – tags (pointers to the reservation stations that will produce the operands)
- Eliminates the need to fetch/write operands from/to registers
  - don't have to write results back to registers, which are immediately read by another instruction
  - implements register renaming
  - performs the same function as forwarding (short-circuiting)

45

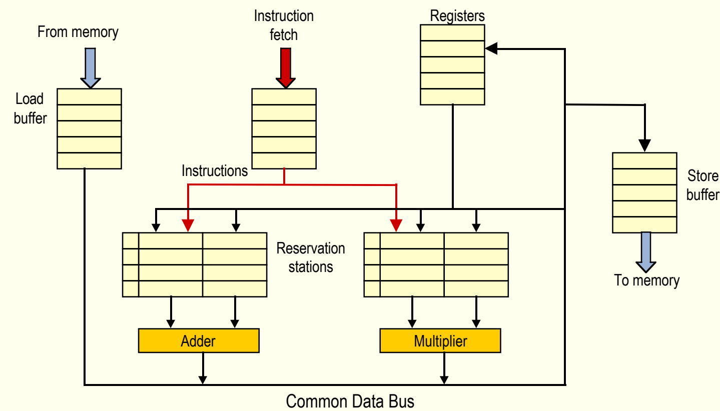
Reservation stations (cont.)

- Reservation stations, functional units, load and store buffers are connected by a Common Data Bus (CDB)
  - memory access (load/store) are treated as functional units
- When the operands of an instruction are available, the instruction can be sent to a functional unit for execution
- Results of execution are broadcasted on the CDB
- Reservation stations listen to the CDB for operand values
  - if a matching tag is seen on the bus, the RS copies the value into its operand field
  - all reservation stations waiting for the value are updated at the same time



46

Organization of Tomasulo's algorithm



47

Data structures in Tomasulos algorithm

- Have to store data describing the state of instructions in reservation stations, registers and load/store buffers
- Tags identify entries in reservation stations
  - used as names for an extended set of registers
  - points to the reservation station that will produce a result needed as an operand
- Issued instructions refer to the operands by tag values
  - not by the registers
- Registers need one additional field
  - the tag of the reservation station that will produce the result to be stored in this register
  - if zero, no currently active instruction is computing a result destined for this register

48

Data structures (cont.)

- Reservation stations have 6 fields
  - *op* – the operation to perform on source operands S_1 and S_2
  - Q_j, Q_k – the tag of the reservation station that will produce the corresponding source operand. A value of zero indicates that the source operand is already available in V_j or V_k, or is not needed
  - V_j, V_k – the value of the source operands. Only one of the *V* and *Q* fields is valid for each operand.
  - *busy* – indicates that the reservation station and its functional unit are occupied

op	Q_j	Q_k	V_j	V_k	*b*

Stages in Tomasulo's algorithm

- Issue
  - get instruction from instruction queue
  - get a free reservation station
  - assign instruction and fetch operands from register if they are available
- Execution
  - if operands are ready, dispatch the instruction to the functional unit for execution
  - if operands are not ready, wait for operands on the CDB
- Write result
  - after an instruction is executed, broadcast the result on the CDB
  - mark the reservation station holding this instruction as free