

The processor

From the instruction set to the processor architecture

1

Resources and technologies

2

An example of instructions

The memory-reference instructions:

- *load register unscaled* (LDUR), and
- *store register unscaled* (STUR)
- LDUR X1,[X2,offset_value] or STUR X1,[X2,offset_value]

The arithmetic-logical instructions

- ADD,SUB,AND, and ORR
- ADD X1, X2, X3

The branch instructions

- compare and branch on zero (CBZ), and
- branch (B)

3

Instruction formats (I)

Field	opcode	Rm	shamt	Rn	Rd
Bit positions	31:21	20:16	15:10	9:5	4:0

The format for ADD,SUB, AND, and ORR. They have three register operands: Rn, Rm, and Rd.

Fields Rn and Rm are sources, and Rd is the destination.

ALUOp					
Field	1986 or 1984	address	0	Rn	Rt
Bit positions	31:21	20:12	11:10	9:5	4:0

b. Load or store instruction

The register Rn is the base register that is added to the 9-bit address field to form the memory address.

- For loads, Rt is the destination register for the loaded value.
- For stores, Rt is the source register whose value should be stored into memory.

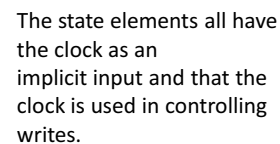
4

6

c. Conditional branch instruction

The register R_t is the source register that is tested for zero. The 19-bit address field is sign-extended, shifted, and added to the

- ## The design



Instruction execution

7

- 1. Fetch instruction from memory.
- 2. Read registers and decode the instruction.
- 3. Execute the operation or calculate an address.
- 4. Access an operand in data memory (if necessary).
- 5. Write the result into a register (if necessary).

8

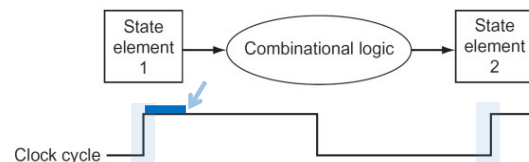
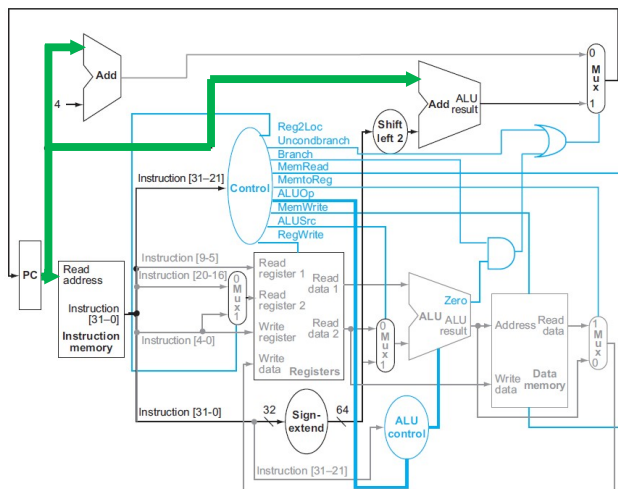
ADD X1,X2,X3

Although everything occurs in one clock cycle, we can think of four steps to execute the instruction:

1. The instruction is fetched, and the PC is incremented.
2. Two registers, X2 and X3, are read from the register file; also, the main control unit computes the setting of the control lines during this step.
3. The ALU operates on the data read from the register file, using portions of the opcode to generate the ALU function.
4. The result from the ALU is written into the destination register (X1) in the register file.

9

ADD X1, X2, X3

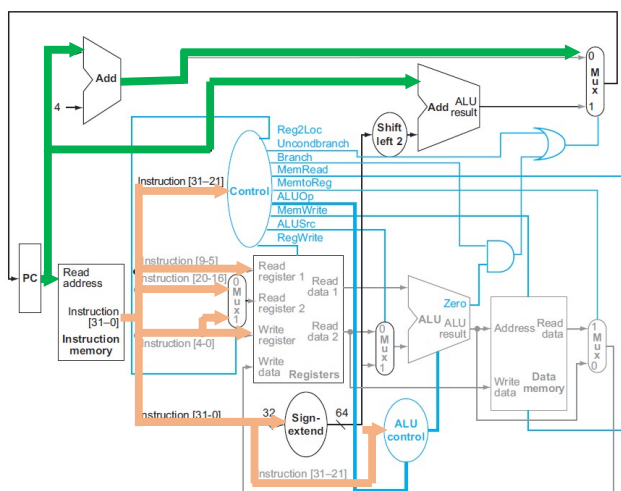


Fetch of the instruction

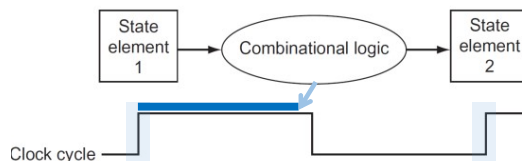
- 1. Send the *program counter* (PC):
 - to the memory that contains the code, and
 - to the two adders used to compute the address of the next instruction.

10

ADD X1, X2, X3



11



1. Fetch of the instruction

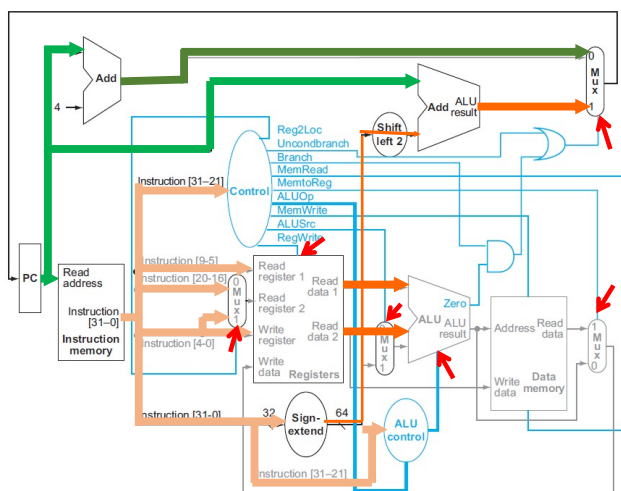
a. Send the *program counter* (PC):

- to the memory that contains the code, and
- to the two adders used to compute the address of the next instruction.

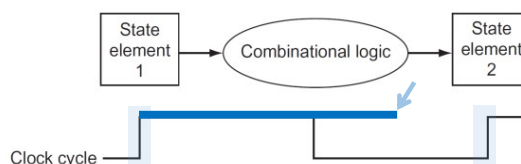
b. The received instruction is sent:

- to the register file, and
- to the control unit.

ADD X1, X2, X3



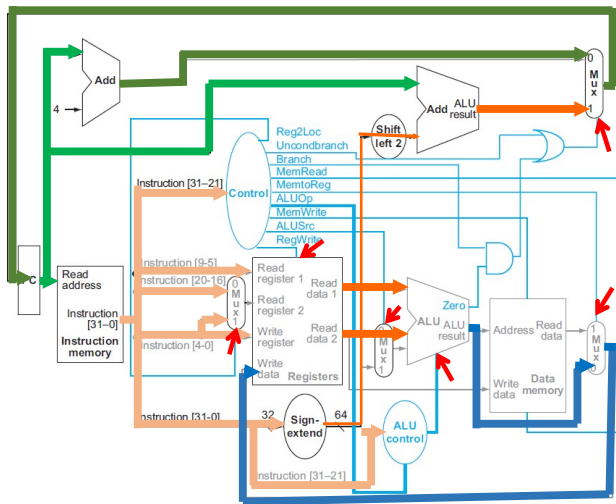
12



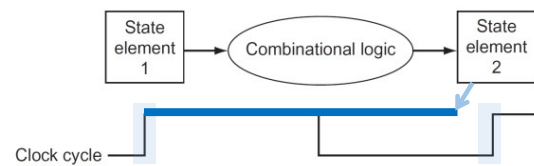
1. Fetch of the instruction

2. Read registers and decode the instruction

ADD X1, X2, X3

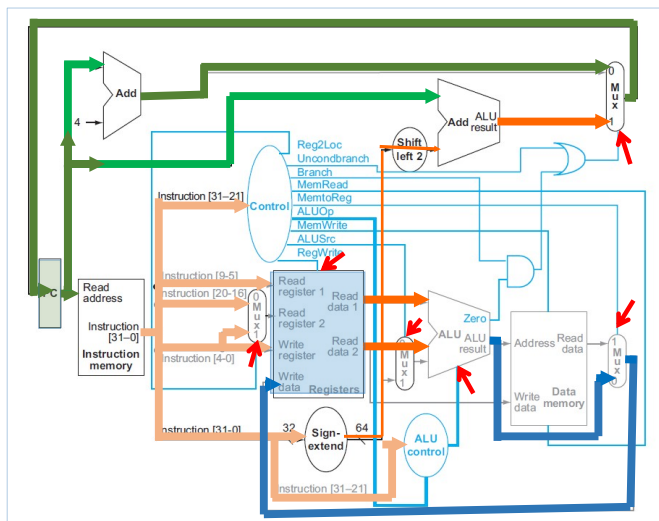


13

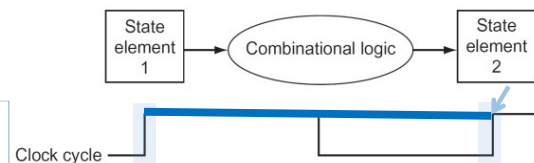


1. Fetch of the instruction and increment PC
2. Read registers and decode the instruction.
3. The ALU operates on the data read from the register file, using portions of the opcode to generate the ALU function.

ADD X1, X2, X3



14



1. Fetch of the instruction
2. Read registers and decode the instruction.
3. The ALU operates on the data read from the register file, using the opcode to generate the ALU function.
4. The result from the ALU is written into the destination register (X1).

The state elements all have the clock as an implicit input and that the clock is used in controlling writes.

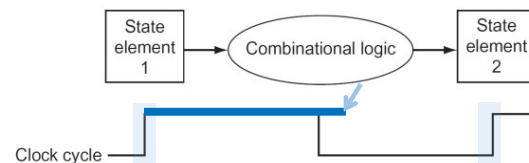
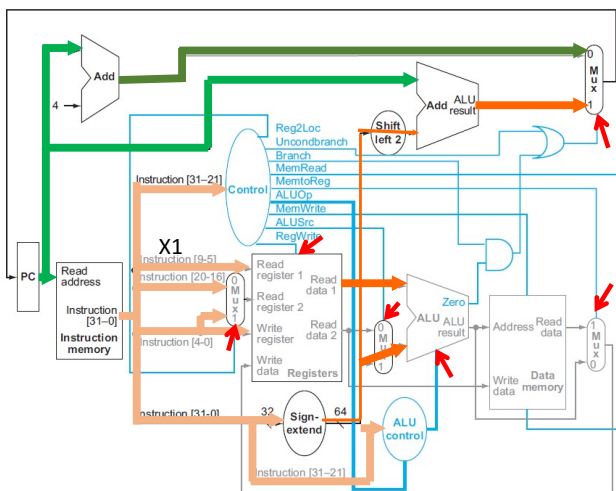
LDUR X1, [X2,offset]

We can think of a load instruction as operating in five steps:

1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. A register (X2) value is read from the register file.
3. The ALU computes the sum of the value read from the register file and the sign-extended 9 bits of the instruction (offset).
4. The sum from the ALU is used as the address for the data memory.
5. The data from the memory unit is written into the register file (X1).

15

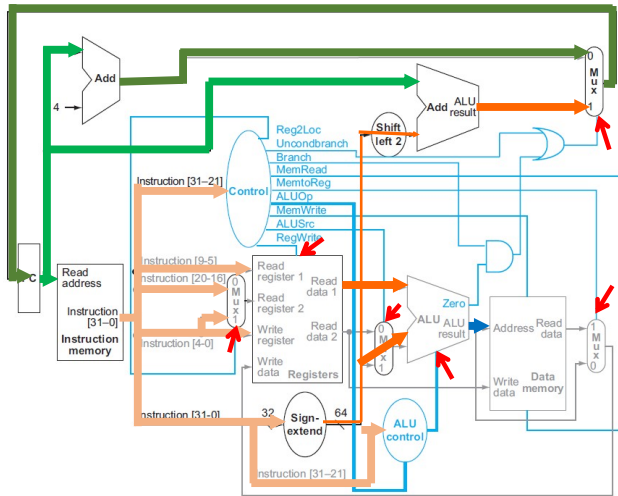
LDUR X1, [X2,offset]



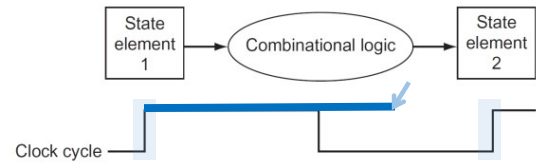
1. Fetch of the instruction and increment PC
2. Read registers and decode the instruction
3. The ALU computes the sum of the value read from the register file and the sign-extended 9 bits of the instruction (offset).

16

LDUR X1, [X2,offset]

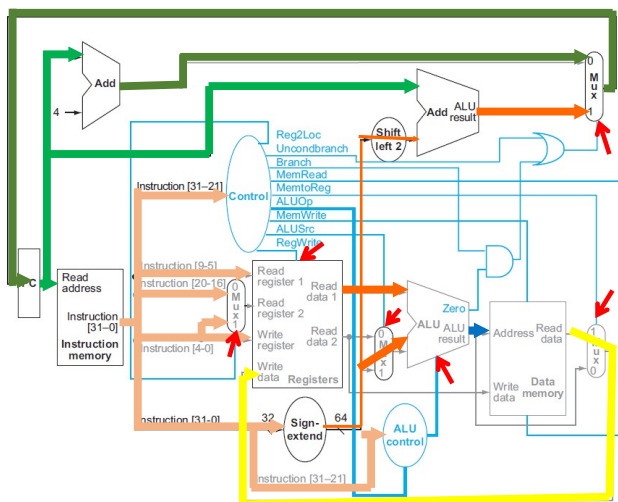


17

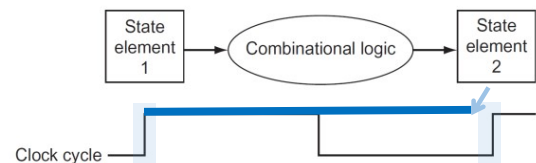


1. Fetch of the instruction and increment PC
2. Read registers and decode the instruction
3. The ALU computes the sum of the value read from the register file and the sign-extended 9 bits of the instruction (offset).
4. The sum from the ALU is used as the address for the data memory.

LDUR X1, [X2,offset]

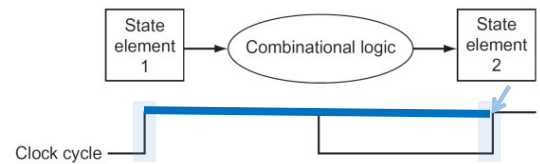
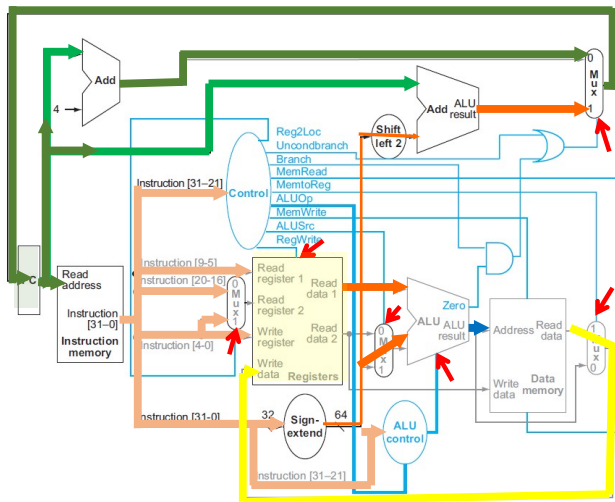


18



1. Fetch of the instruction and increment PC
2. Read registers and decode the instruction
3. The ALU computes the sum of the value read from the register file and the sign-extended 9 bits of the instruction (offset).
4. The sum from the ALU is used as the address for the data memory.

LDUR X1, [X2,offset]



1. Fetch of the instruction and increment PC
2. Read registers and decode the instruction
3. The ALU computes the sum of the value read from the register file and the sign-extended 9 bits of the instruction (offset).
4. The sum from the ALU is used as the address for the data memory.
5. The data from the memory unit is written into the register file (X1).

The state elements all have the clock as an implicit input and that the clock is used in controlling writes.

19

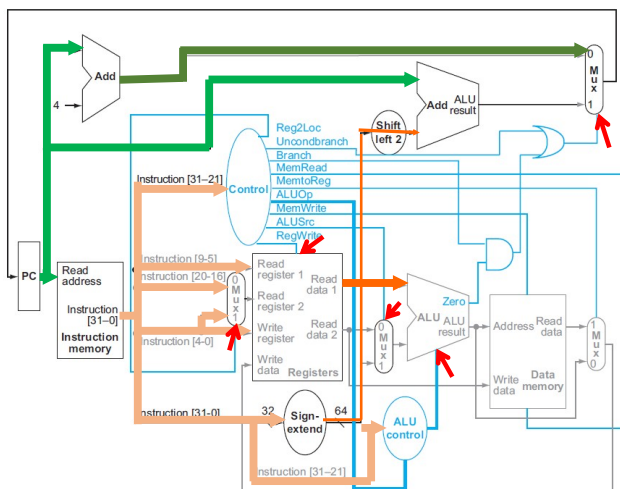
CBZ X1, offset

The four steps in execution:

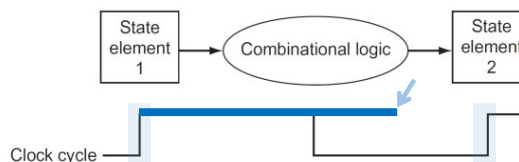
1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. The register, X1 is read from the register file using bits 4:0 of the instruction (Rt).
3. The ALU passes the data value read from the register file. The value of PC is added to the sign-extended, 19 bits of the instruction (offset) are shifted left by two; the result is the branch target address.
4. The Zero status information from the ALU is used to decide which adder result to store in the PC.

20

CBZ X1, offset

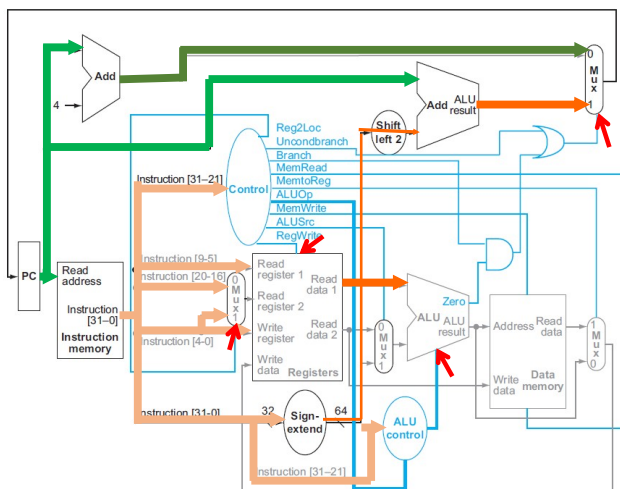


21

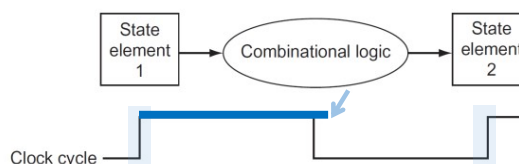


1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. The register, X1 is read from the register file using bits 4:0 of the instruction (Rt).

CBZ X1, offset

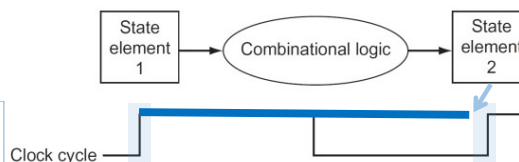
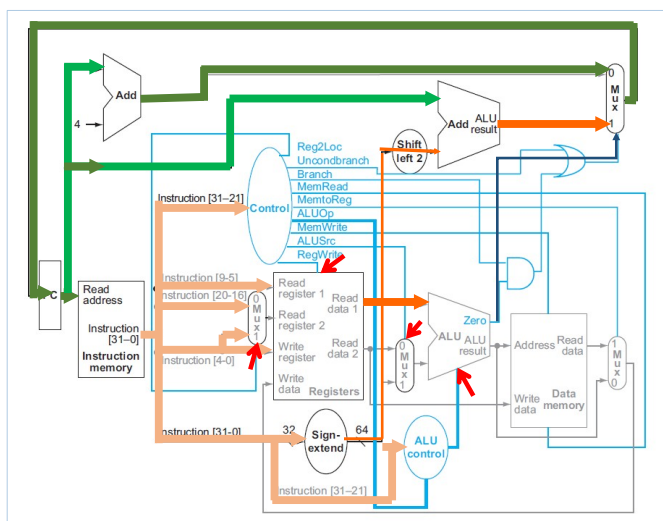


22



1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. The register, X1 is read from the register file using bits 4:0 of the instruction (Rt).
3. The ALU passes the data value read from the register file. The value of PC is added to the sign-extended, 19 bits of the instruction (offset) are shifted left by two; the result is the branch target address.

CBZ X1, offset

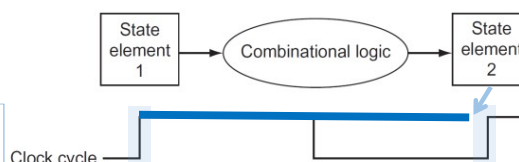
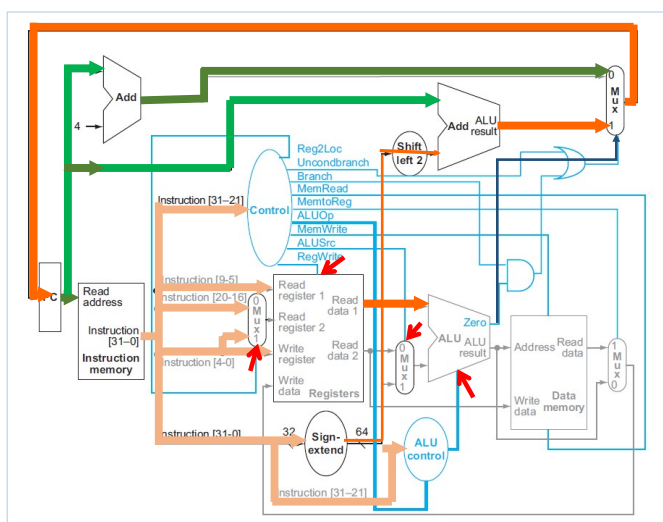
No jump
 $X1 \neq 0$ 

1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. The register, X1 is read from the register file using bits 4:0 of the instruction (Rt).
3. The ALU passes the data value read from the register file. The value of PC is added to the sign-extended, 19 bits of the instruction (offset) are shifted left by two; the result is the branch target address.
4. The Zero status information from the ALU is used to decide which adder result to store in the PC.

The state elements all have the clock as an implicit input and that the clock is used in controlling writes.

23

CBZ X1, offset

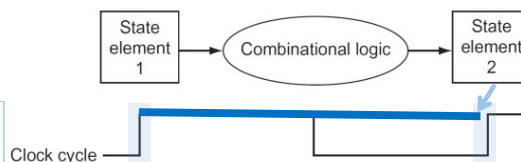
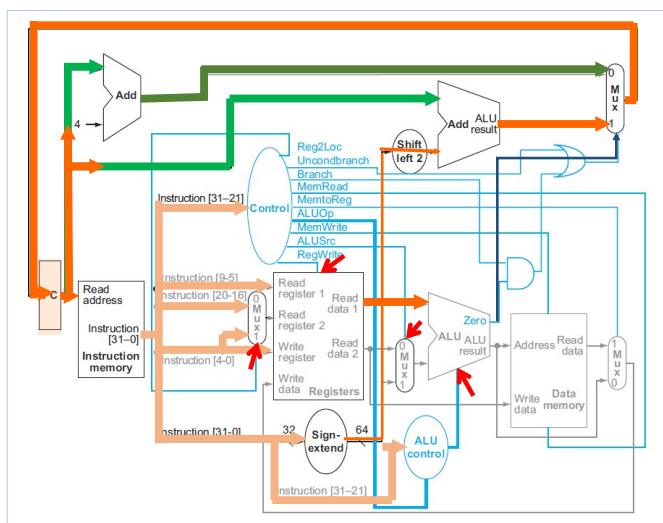
Jump
 $X1 = 0$ 

1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. The register, X1 is read from the register file using bits 4:0 of the instruction (Rt).
3. The ALU passes the data value read from the register file. The value of PC is added to the sign-extended, 19 bits of the instruction (offset) are shifted left by two; the result is the branch target address.
4. The Zero status information from the ALU is used to decide which adder result to store in the PC.

The state elements all have the clock as an implicit input and that the clock is used in controlling writes.

24

CBZ X1, offset

Jump
X1 = 0

1. An instruction is fetched from the instruction memory, and the PC is incremented.
2. The register, X1 is read from the register file using bits 4:0 of the instruction (Rt).
3. The ALU passes the data value read from the register file. The value of PC is added to the sign-extended, 19 bits of the instruction (offset) are shifted left by two; the result is the branch target address.
4. The Zero status information from the ALU is used to decide which adder result to store in the PC.

The state elements all have the clock as an implicit input and that the clock is used in controlling writes.

25

Why a Single-Cycle Implementation is not Used

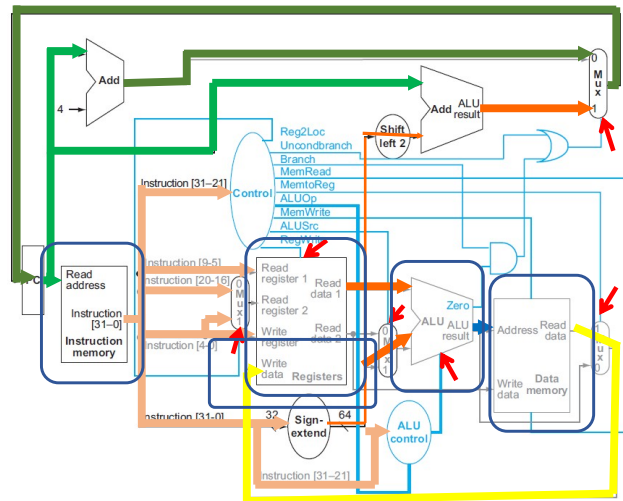
Historically, early computers with very simple instruction sets did use this implementation technique.

26

The result: the CPI is 1 (1)

The longest possible path in the processor determines the clock cycle.

- LDUR X1,[X2,offset_value] or STUR X1,[X2,offset_value]
Uses **5** functional units in series:
 • the instruction memory,
 • the register file,
 • the ALU,
 • the data memory, and
 • the register file

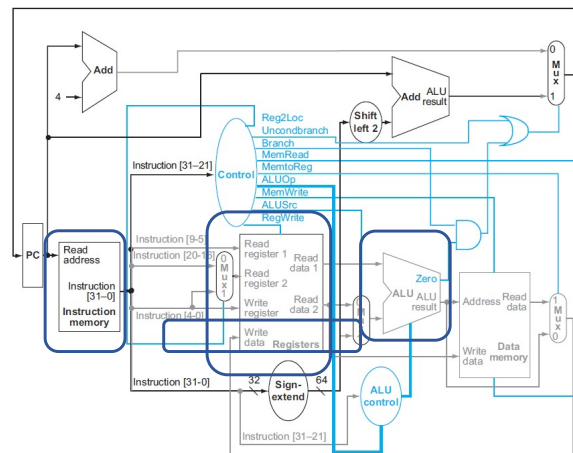


27

The result: the CPI is 1 (1)

The longest possible path in the processor determines the clock cycle.

- LDUR X1,[X2,offset_value] or STUR X1,[X2,offset_value]
 • Uses **5** functional units in series the instruction memory, the register file, the ALU, the data memory, and the register file
- ADD X1, X2, X3
 Uses **4** functional units in series the instruction memory,
 • the register file,
 • the ALU, and
 • the register file



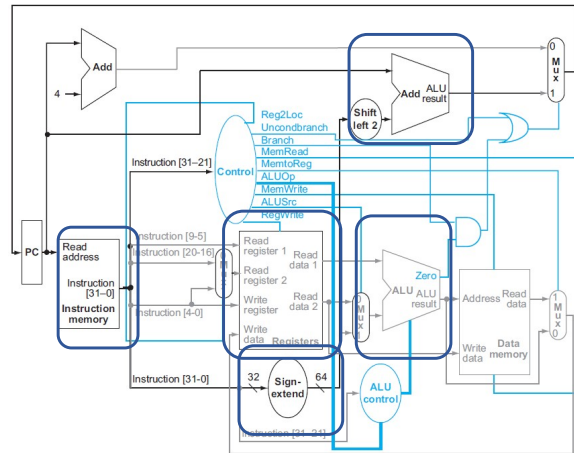
28

The result: the CPI is 1 (1)

The longest possible path in the processor determines the clock cycle.

- LDUR X1,[X2,offset_value] or STUR X1,[X2,offset_value]
 - Uses 5 functional units in series the instruction memory, the register file, the ALU, the data memory, and the register file
- ADD X1, X2, X3
 - Uses 4 functional units in series the instruction memory, the register file, the ALU, and the register file
- CBZ X1, offset
compare and branch on zero (CBZ).
Uses 4 functional units in series
 - the instruction memory,
 - The register file,
 - the sign-extended, and
 - the ALU.

Although the CPI is 1, the overall performance of a single-cycle implementation is likely to be poor, since the clock cycle is too long.

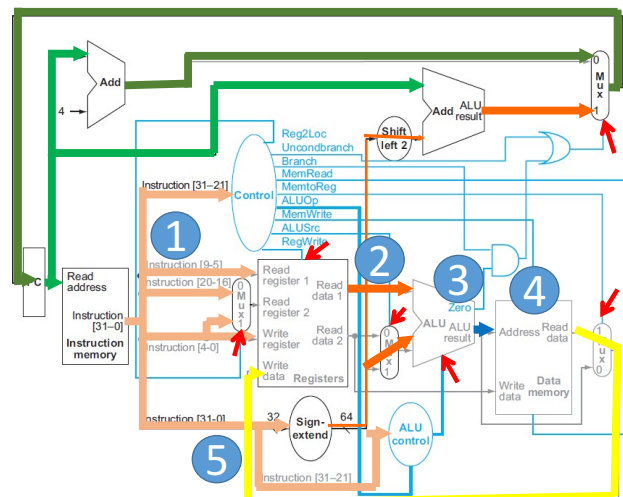


29

Pipelining

1. Fetch instruction from memory.
2. Read registers and decode the instruction.
3. Execute the operation or calculate an address.
4. Access an operand in data memory (if necessary).
5. Write the result into a register (if necessary).

LDUR X1, [X2,offset]



30

Total time for each instruction (I)

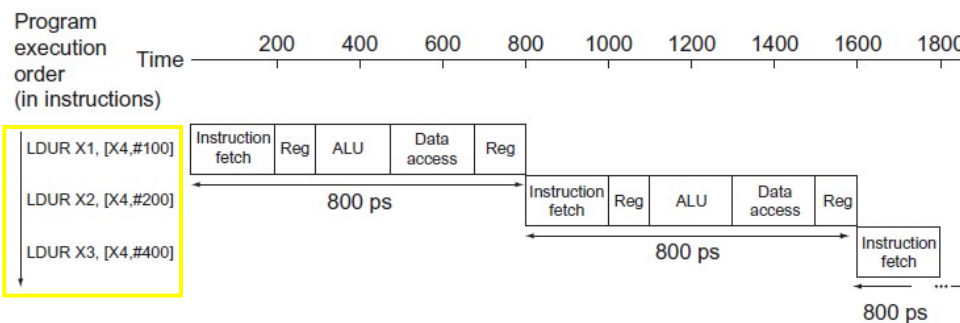
The longest possible path in the processor determines the clock cycle.

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load register (LDUR)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store register (STUR)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (ADD, SUB, AND, ORR)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (CBZ)	200 ps	100 ps	200 ps			500 ps

31

Total time for each instruction (II)

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load register (LDUR)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store register (STUR)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (ADD, SUB, AND, ORR)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (CBZ)	200 ps	100 ps	200 ps			500 ps

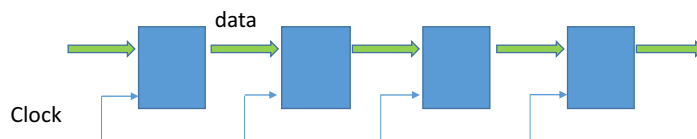


32

Pipeline data transfer

Synchronous method

- Synchronous method



The timing signal (clock) causes all output of units to be transferred to the next units.

The timing signal occurs at fixed intervals, taking into account the slowest unit.

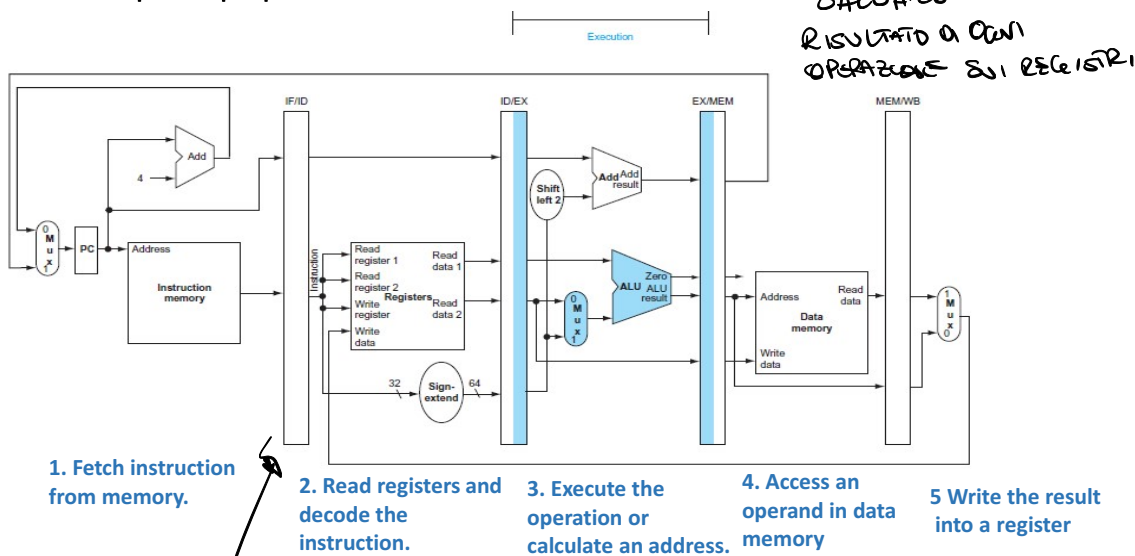
Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load register (LDUR)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store register (STUR)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (ADD, SUB, AND, ORR)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (CBZ)	200 ps	100 ps	200 ps			500 ps

200 ps, for clock cycle

33

33

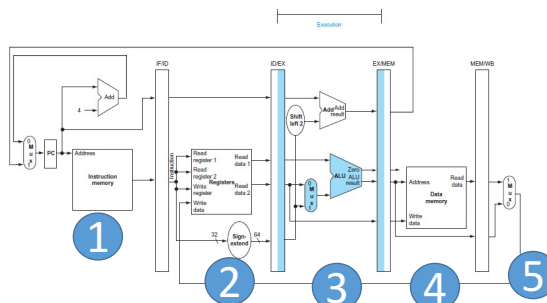
Simple pipeline solution



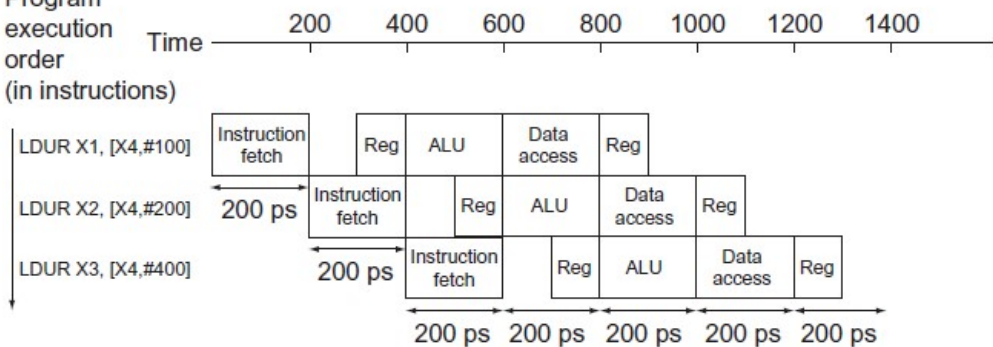
34

Simple pipeline solution (1)

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load register (LDUR)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store register (STUR)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (ADD, SUB, AND, ORR)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (CBZ)	200 ps	100 ps	200 ps			500 ps

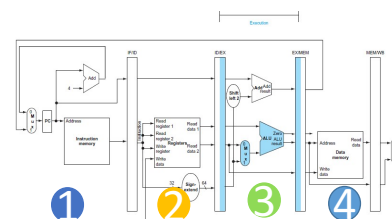
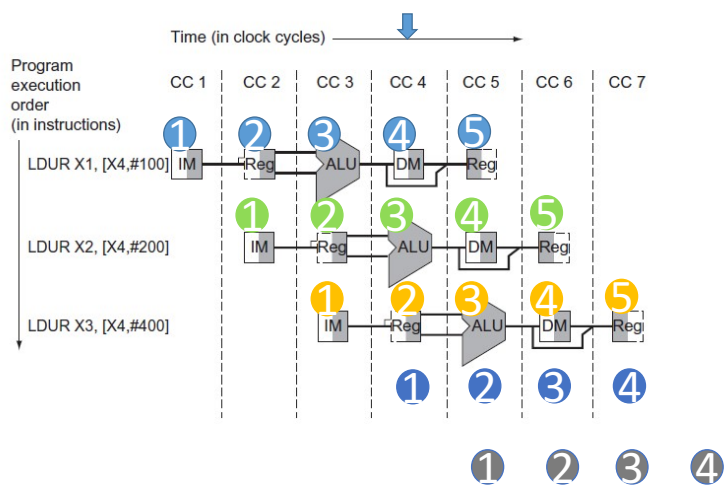


Program
execution
order
(in instructions)



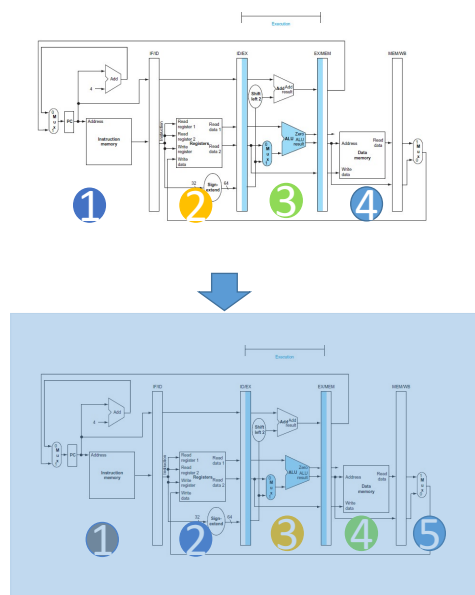
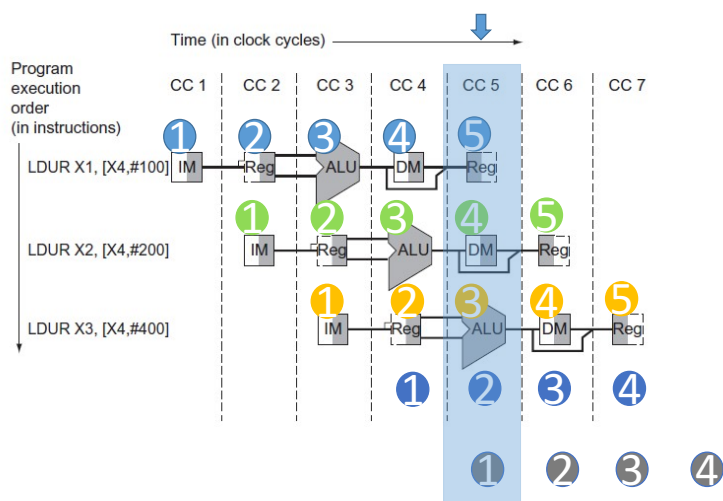
35

Simple pipeline solution (2)



36

Simple pipeline solution (3)



37

Speedup

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load register (LDUR)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store register (STUR)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (ADD, SUB, AND, ORR)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (CBZ)	200 ps	100 ps	200 ps			500 ps

Pipelining improves performance by *increasing instruction throughput*,
Speedup => 4

In contrast to increasing the execution time of an individual instruction,
 LDUR X1, [X2, offset] 800ps => 1000ps

Instruction throughput is the important metric because real programs execute billions of instructions.

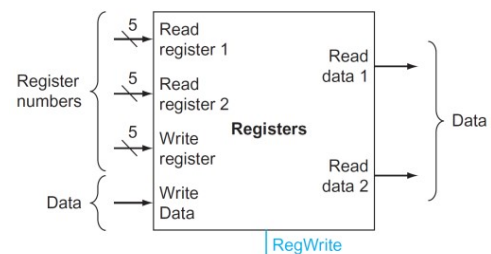
38

The multi-port register file

39

The multi-port register file

- The multiported register file contains all the registers and has two read ports and one write port.
- The register file always outputs the contents of the registers corresponding to the Read register inputs on the outputs; no other control inputs are needed.
- A register write must be explicitly indicated by asserting the write control signal.
- The writes are edge-triggered, so that all the write inputs must be valid at the clock edge.
- Since writes to the register file are edge-triggered, our design can legally read and write the same register within a clock cycle: the read will get the value written in an earlier clock cycle, while the value written will be available to a read in a subsequent clock cycle.

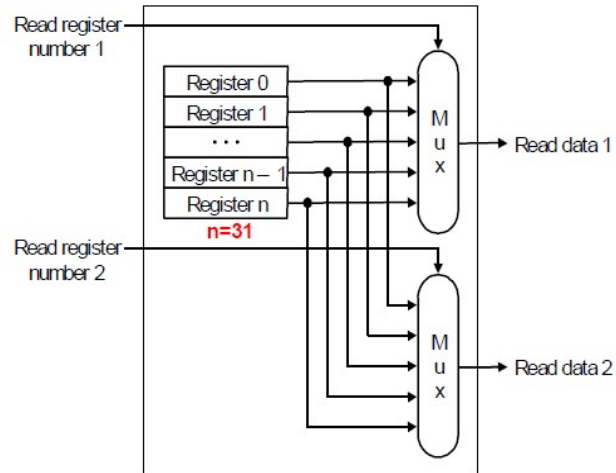


40

Read part

The multiported register file contains two read ports.

The register file always outputs the contents of the registers corresponding to the Read register inputs on the outputs; no other control inputs are needed.



41

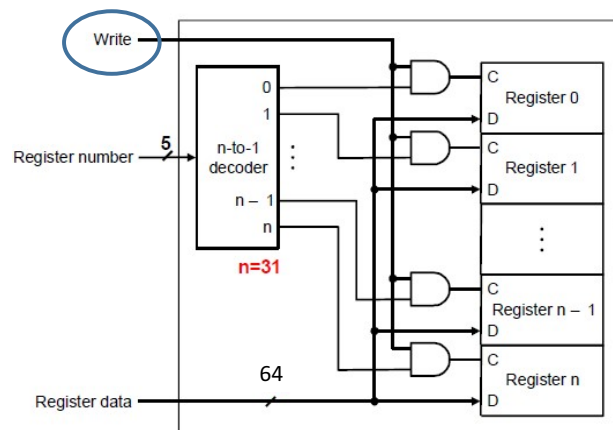
Write part

A register write must be explicitly indicated by asserting the write control signal.

The writes are edge-triggered, so that all the write inputs must be valid at the clock edge.

Since writes to the register file are edge-triggered, our design can read and write the same register within a clock cycle.

- The read will get the value written in an earlier clock cycle,
- the value written will be available to a read in a subsequent clock cycle.



42

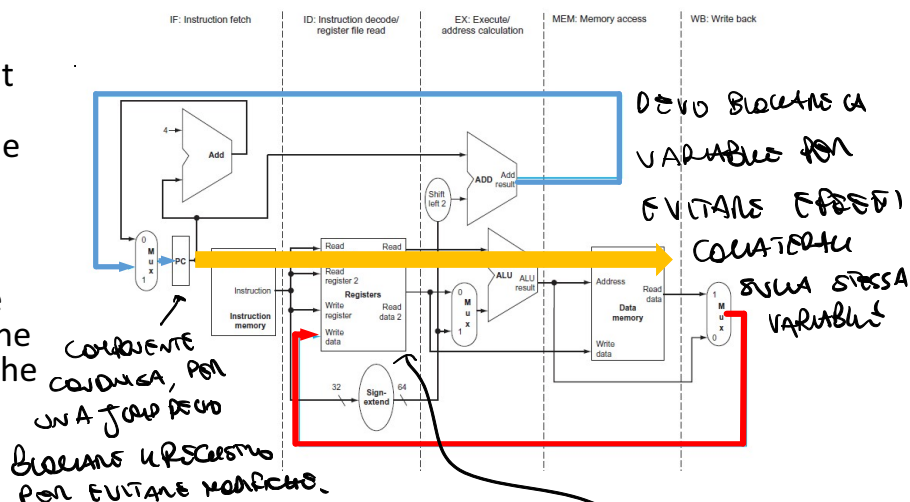
Pipeline Hazards

43

① POSSO PULIRE UN COCCO SUI REGISTRI ② POSSO IDENTIFICARE
MA FACENDO IL
VEDO DI ADETTORIANE
WOLFINO

Two exceptions to left-to-right flow of instructions

- The write-back stage, which places the result back into the register file in the middle of the datapath.
- The selection of the next value of the PC, choosing between the incremented PC and the branch address from the MEM stage.



44

DAL PUNTO DI VISTA DEL SW: IL PROBLEMA E' CHE SE E' UN DATO
GENERATO SU CUI AL PROCESSI POSSONO
LAVORARE
↓
CI VOGLIE LA SINCRONIZZAZIONE
(SEMPRE ESSENZIALE)

Structural Hazard

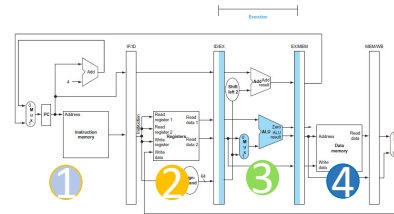
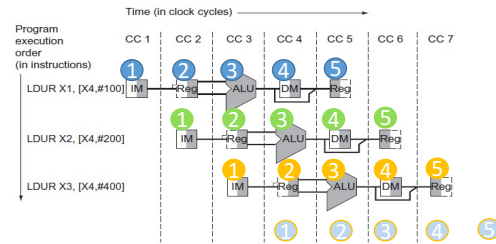
The hardware (bus, ALU, memories, ...) cannot support the combination of instructions that we want to execute in the same clock cycle.

Suppose that we had a single memory instead of two memories.

If the pipeline had a fourth instruction, we would see that in the same clock cycle,

- the first instruction is accessing data from memory
- while the fourth instruction is fetching an instruction from that same memory.

Without two memories (Instruction Memory and Data Memory), the pipeline could have a structural hazard.



45

Data Hazards

Data hazards occur when the pipeline must be stalled because one step must wait for another to complete.

For example:

- ADD **X19**, X0, X1 (X19, destination register)
- SUB X2, **X19**, X3 (X19, source register)

Without intervention, a data hazard could severely stall the pipeline.

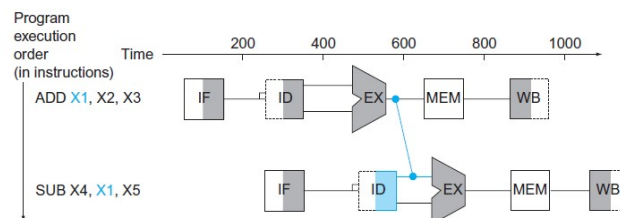
The add instruction doesn't write its result until the fifth stage, meaning that we would have to waste three clock cycles in the pipeline.

46

Data Hazards: solutions

forwarding or bypassing

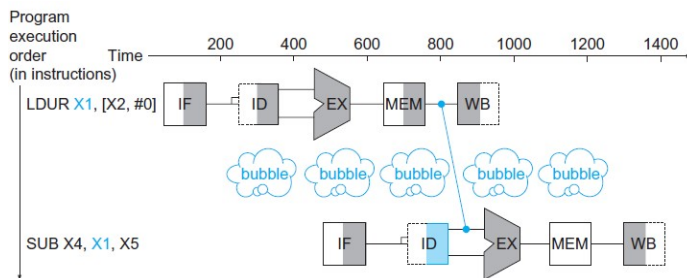
- The primary solution is based on the observation that we don't need to wait for the instruction to complete before trying to resolve the data hazard.
- As soon as the ALU creates the sum for the add, we can supply it as an input for the subtract.
- Adding extra hardware to retrieve the missing item early from the internal resources is called **forwarding** or **bypassing**.



47

Load and arithmetic operations

A pipeline stall, often given the nickname bubble



Even with forwarding, we would have to stall one stage for a **load-use data hazard**.

48

Reordering Code to Avoid Pipeline Stalls

- Consider the following code segment in C:
- $a = b + e;$
- $c = b + f;$

*POSSIBILI LAVORARE
SOLLO SUL SW*

Assuming all variables are in memory and are addressable as offsets from X0:

```
LDUR X1, [X0,#0]      // Load b
LDUR X2, [X0,#8]      // Load e
ADD X3, X1,X2          // b + e
STUR X3, [X0,#24]     // Store a
LDUR X4, [X0,#16]     // Load f
ADD X5, X1,X4          // b + f
STUR X5, [X0,#32]     // Store c
```

```
LDUR X1, [X0,#0]
LDUR X2, [X0,#8]
LDUR X4, [X0,#16]
ADD X3, X1,X2
STUR X3, [X0,#24]
ADD X5, X1,X4
STUR X5, [X0,#32]
```

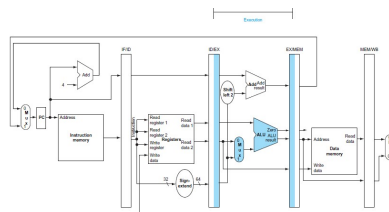
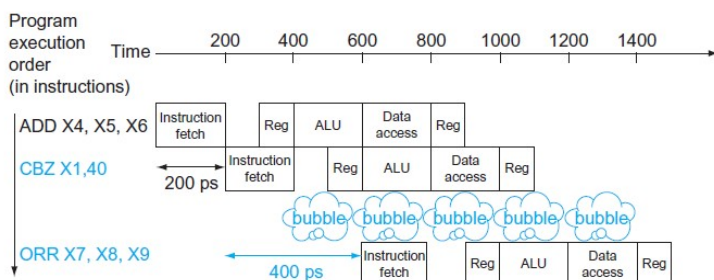
*ANTICIPARE
LEAD*

*NON POSSO RIVEDERE
L'OPERAZIONE CORRE
VOCALICO AN CHE
C'È UN BUBBLE
FUGA*

49

Control Hazards, also called branch hazard

Control hazard, arising from the need to decide based on the results of one instruction while others are executing.

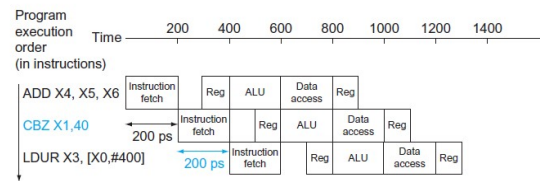


50

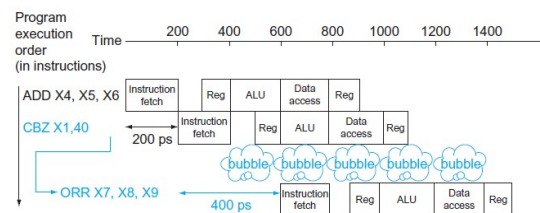
Prediction to handle conditional branches

One simple approach is to predict always that conditional branches will be untaken.

- The pipeline proceeds at full speed.



- Only when conditional branches are taken does the pipeline stall.



51

branch prediction

- One popular approach to the dynamic prediction of conditional branches is keeping history for each conditional branch as taken or untaken, and then using the recent past behavior to predict the future.
- Dynamic branch predictors can correctly predict conditional branches with more than 90% accuracy.
- When the guess is wrong, the pipeline control must ensure that the instructions following the wrongly guessed conditional branch have no effect and must restart the pipeline from the proper branch address.

52

dynamic branch prediction

With deeper pipelines, the branch penalty increases when measured in **clock cycles**.

With multiple issue, the branch penalty increases in terms of **instructions lost**.

One approach is to look up the address of the instruction to see if the conditional branch was taken the last time this instruction was executed, and, if so, **to begin fetching new instructions from the same place as the last time**.

One implementation of that approach is a **branch prediction buffer** or **branch history** table.

A branch prediction buffer is a small memory indexed by the lower portion of the address of the branch instruction.

The memory contains a bit that says whether the branch was recently taken or not.

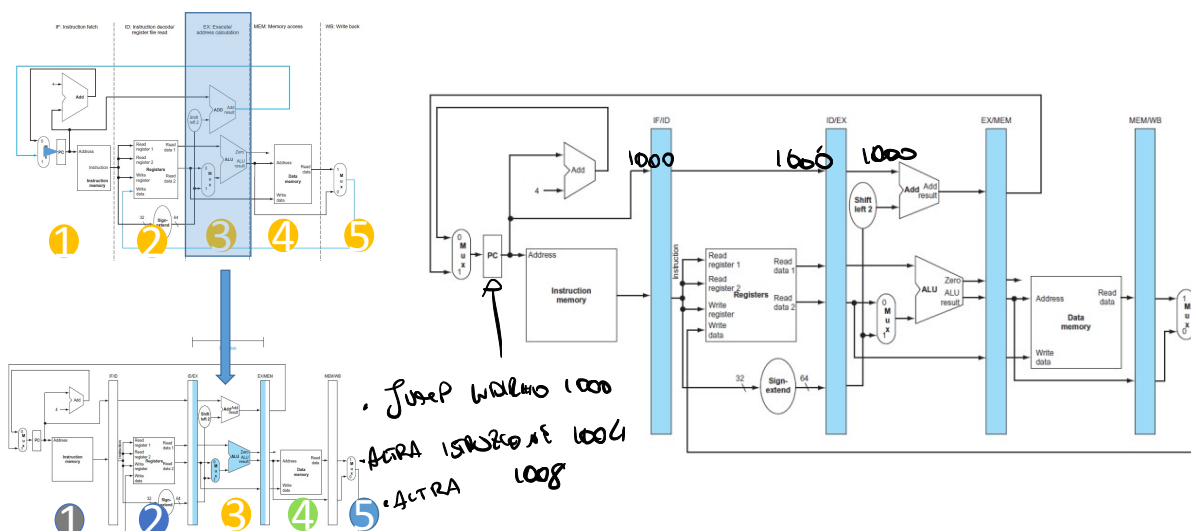
If the hint turns out to be wrong, the incorrectly predicted instructions are deleted, the prediction bit is inverted and stored back, and the proper sequence is fetched and executed.

53

Pipelined Datapath and Control

54

Pipeline registers



55

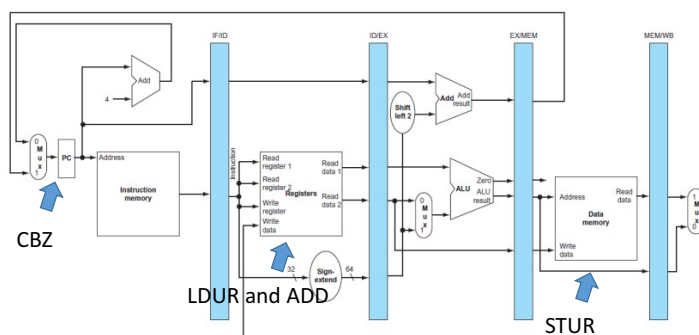
Pipeline registers

All instructions advance during each clock cycle from one pipeline register to the next.

There is no pipeline register at the end of the write-back stage.

All instructions must update some state in the processor: the register file, memory, or the PC.

Every instruction updates the PC, whether by incrementing it or by setting it to a branch destination address



56

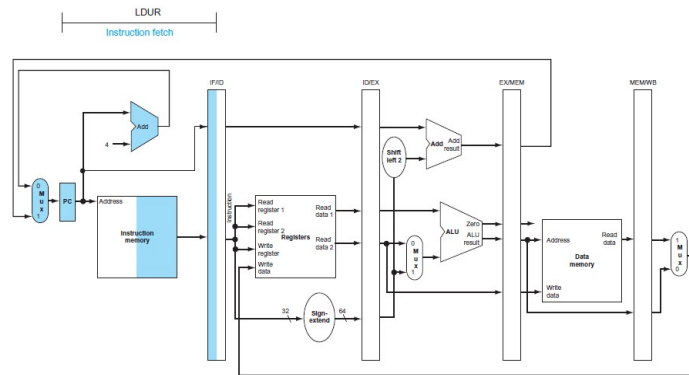
LDUR, *Instruction fetch*

The computer cannot know which type of instruction is being fetched, so it must prepare for any instruction, passing potentially needed information down the pipeline.

The **instruction being read** from memory using the address in the PC and then being **placed in the IF/ID pipeline register**.

The **PC address is incremented by 4** and then **written back into the PC** to be ready for the next clock cycle.

The **incremented address is also saved in the IF/ID pipeline register** in case it is needed later for an instruction, such as **CBZ**.

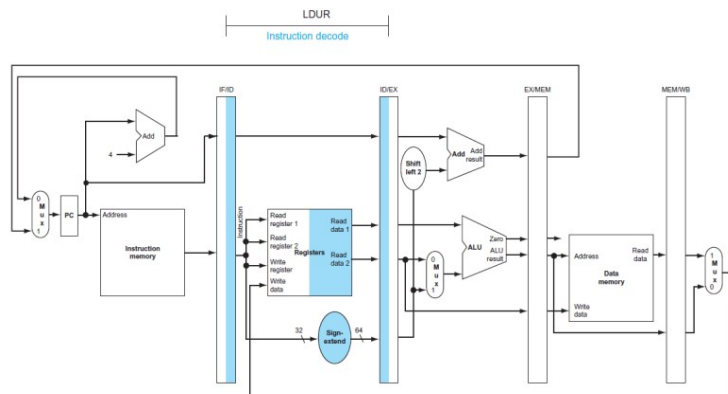


57

LDUR, *Instruction decode and register file read*

The IF/ID pipeline register supplies the immediate field, which is sign-extended to 64 bits, and the **register numbers** to read the two registers.

All three values are stored in the ID/EX pipeline register, along with the incremented PC address.

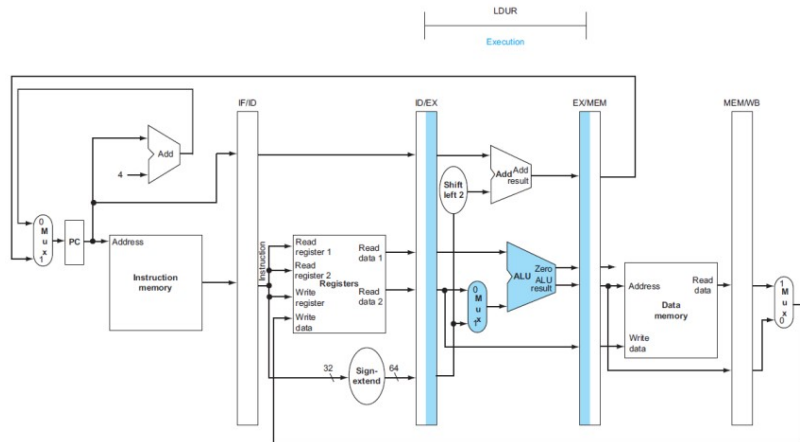


58

LDUR, address calculation

The load instruction **reads the contents of a register** and the **sign-extended immediate from the ID/EX pipeline register** and **adds them using the ALU**.

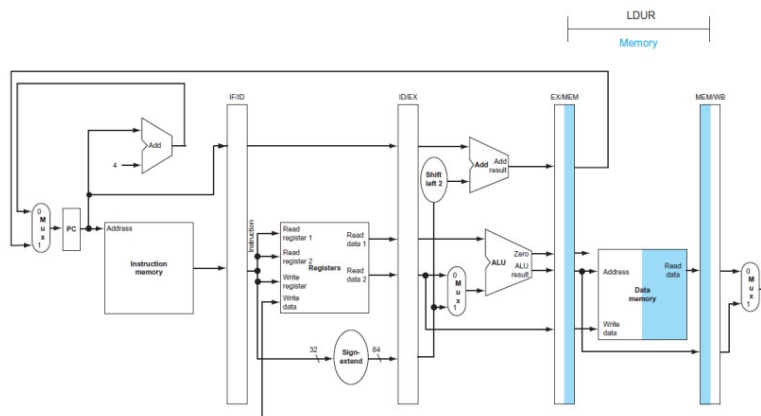
That **sum** is placed in the **EX/MEM pipeline register**.



59

LDUR, Memory access

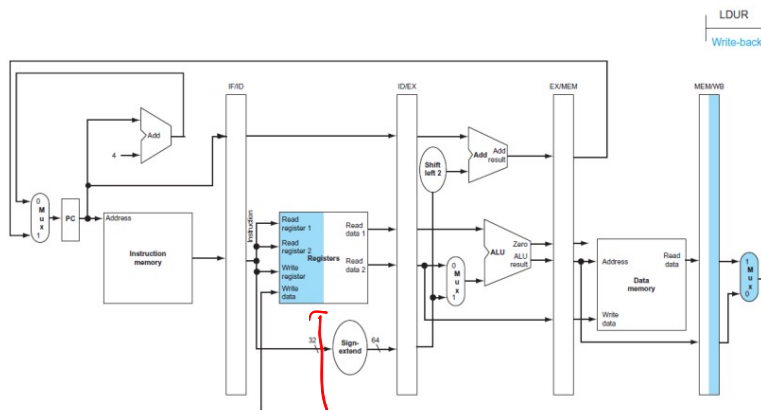
- The load instruction **reads the data memory** using the **address from the EX/MEM pipeline register** and loads the **data** into the **MEM/WB pipeline register**.



60

LDUR, Write-back

The final step:
reading the data
 from the MEM/WB
pipeline register and
writing it into the
 register file.



new corrected
 C'E W'ACRA (M)KORF Qui

61

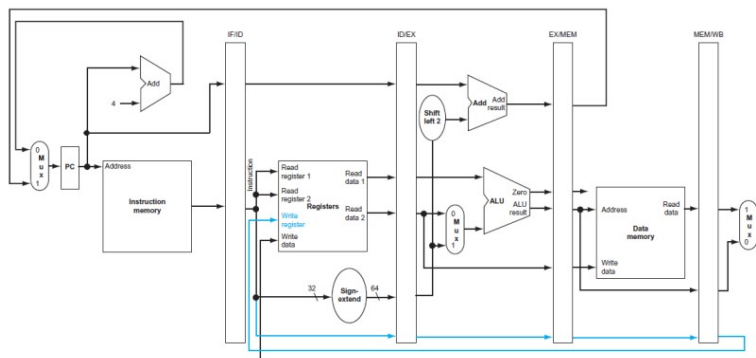
LDUR, Write-back

The corrected pipelined datapath to handle the load instruction properly

The final step: **reading the data** from the MEM/WB pipeline register and **writing** it into the register file.

We need to preserve the destination register number in the load instruction.

Just as store passed the register *value* from the ID/EX to the EX/MEM pipeline registers for use in the MEM stage, load must pass the register *number* from the ID/EX through EX/MEM to the MEM/WB pipeline register for use in the WB stage.



62

Pipelined Control for LDUR instruction

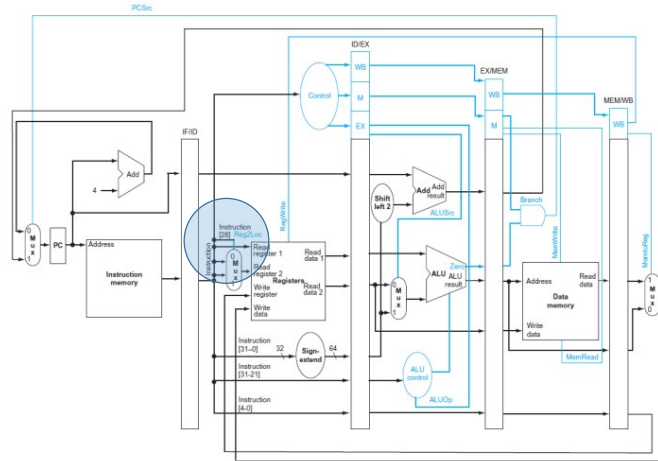
- We assume that the PC is written on each clock cycle, so there is no separate write signal for the PC.
- The pipeline registers are written during each clock cycle. There are no separate write signals for the pipeline registers (IF/ID, ID/EX, EX/MEM, and MEM/WB).

Because each control line is associated with a component active in only a single pipeline stage, we can divide the control lines into five groups according to the pipeline stage.

1. **Instruction fetch:** The control signals to read instruction memory and to write the PC are always asserted, so there is **nothing special to control** in this pipeline stage.

2. **Instruction decode/register file read:** We need to select the correct register number for read register 2, so the signal **Reg2Loc** is set.

The signal selects instruction bits 20:16 (Rm) or 4:0 (Rt).



63

Pipelined Control

3. **Execution/address calculation:** The signals to be set are **ALUOp** and **ALUSrc**.

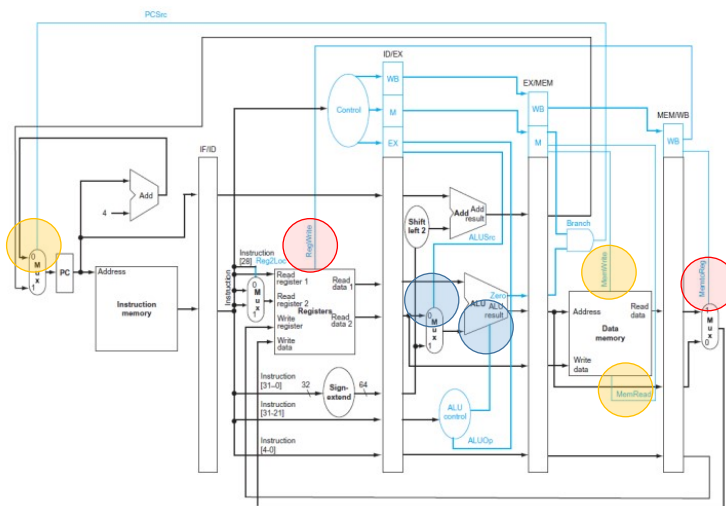
- The signals select the ALU operation and either Read data 2 or a sign-extended immediate as inputs to the ALU.

4. **Memory access:** The control lines set in this stage are **Branch**, **MemRead**, and **MemWrite**.

The compare and branch on zero, load, and store instructions set these signals.

PCsrc selects the next sequential address unless control asserts Branch and the ALU result was 0.

5. **Write-back:** The two control lines are **MemoReg**, which decides between sending the ALU result or the memory value to the register file, and **RegWrite**, which writes the chosen value.



64

Interrupt and Exceptions

65

Exception

Exception to refer to *any* unexpected change in control flow without distinguishing whether the cause is internal or external
Interrupt only when the event is externally caused.

Type of event	From where?	ARMv8 terminology
System reset	External	Exception
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Floating-point arithmetic overflow or underflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

66

How Exceptions are Handled (I)

The basic action that the processor must perform when an exception occurs is to **save the address** of the instruction in the *exception link register* (ELR) and then **transfer control** to the operating system at some specified address.

The operating system **can then take the appropriate action**, which may involve providing some service to the user program, taking some predefined action in response to a malfunction, or stopping the execution of the program and reporting an error.

After performing whatever action is required because of the exception, the **operating system can terminate the program or may continue its execution**, using the ELR to determine where to restart the execution of the program.

For the operating system to handle the exception, it must know:

1. the **reason** for the exception,
2. the address of **instruction** that caused it.

67

How Exceptions are Handled (II)

There are two main methods used to communicate the reason for an exception.

The method that may used is to include a status register which holds a field that indicates the reason for the exception.

- A single entry point for all exceptions may be used, and the operating system decodes the status register to find the cause.

A second method is to use **vectored interrupts**.

- In a vectored interrupt, the address to which control is transferred is determined by the cause of the exception, possibly added to a base register that points to memory range for vectored interrupts.

Exception type	Exception vector address to be added to a Vector Table Base Register
Unknown Reason	00 0000 _{hex}
Floating-point arithmetic exception	10 1100 _{hex}
System Error (hardware malfunction)	10 1111 _{hex}

68

Exceptions in a Pipelined Implementation *as for the taken branch*

A pipelined implementation treats exceptions as another form of control hazard.

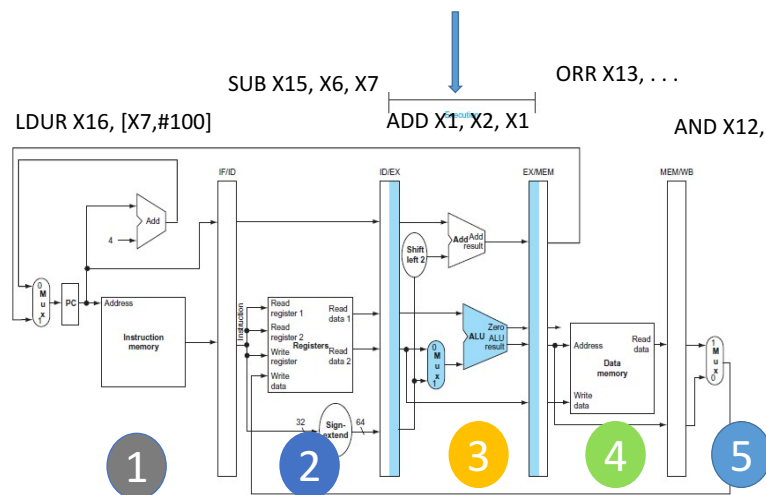
For example, suppose there is a hardware malfunction in an add instruction.

- We must flush the instructions that follow the ADD instruction from the pipeline and
- begin fetching instructions from the new address.

69

Exception occurred while executing ADD

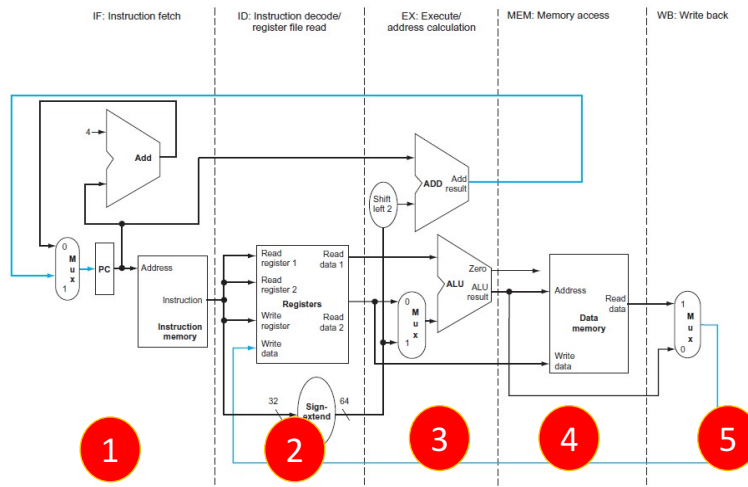
LDUR X16, [X7,#100]
SUB X15, X6, X7
ADD X1, X2, X1
ORR X13, ...
AND X12, .



70

The NOP operation

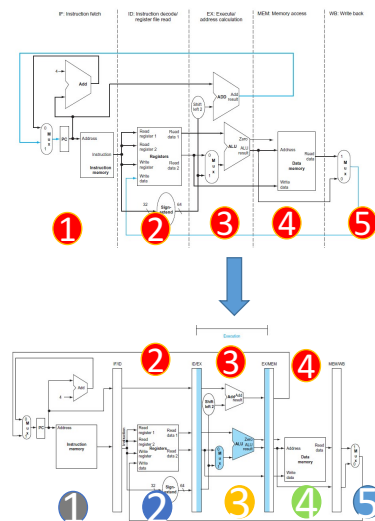
- LDUR X16, [X7,#100]
- SUB X15, X6, X7
- ADD X1, X2, X1
- ORR X13, ...
- AND X12, .



71

The exception and the NOP

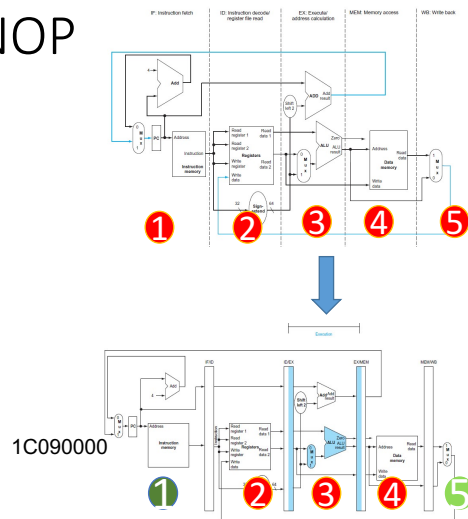
- LDUR X16, [X7,#100]
- SUB X15, X6, X7
- ADD X1, X2, X1
- ORR X13, ...
- AND X12, .



72

The exception and the NOP

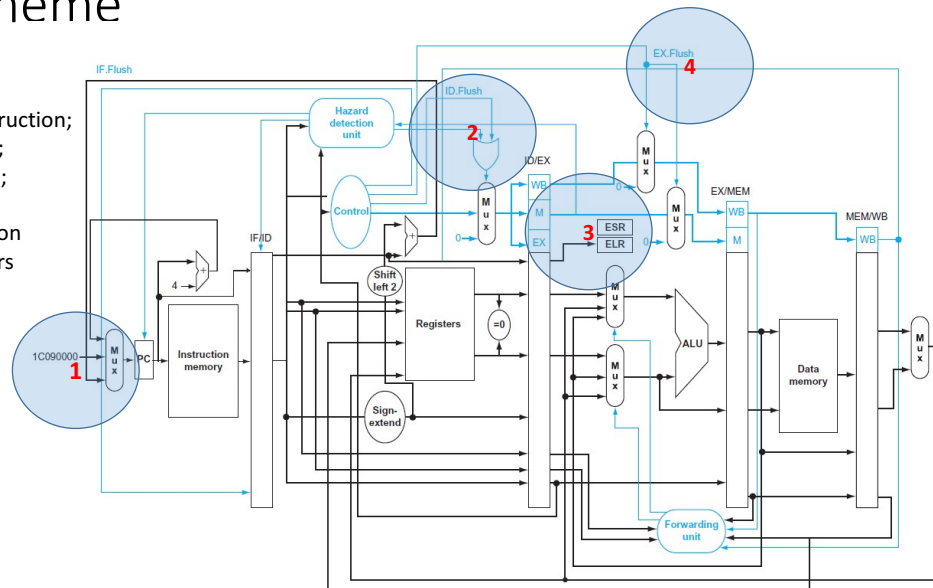
- Fetch instruction from location 0000 0000 1C09 0000hex, which is the address of the first instruction of the exception manager
- Flush instruction in the ID stage
- Flush instruction in the EX phase
- Flush instruction in the MEM phase



73

The final scheme

1. Address of new instruction;
2. Flush the ID section;
3. Flush the EX section;
4. Write the exception information in the status registers



74

Potential parallelism among instructions

Pipelining exploits the potential **parallelism** among instructions.

This parallelism is called, naturally enough, **instruction-level parallelism (ILP)**.

75

Increasing the instruction level parallelism

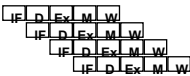
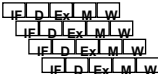
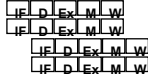


Pipelining exploits the potential **parallelism** among instructions.

Two primary methods for increasing the potential amount of instruction level parallelism:

- **Increasing the pipeline depth.** To overlap more instructions.
- **Multiple issue.** Replicate the internal components of the computer so that it can launch multiple instructions in every pipeline stage.
 - *Launching multiple instructions per stage allows the instruction execution rate to exceed the clock rate or, stated alternatively, the CPI to be less than 1.*
 - *Today's high-end microprocessors attempt to issue from three to six instructions in every clock cycle.*

76

Current techniques to accelerate the single core

° Pipelining		<u>Limitation</u>
° Super-pipeline	<ul style="list-style-type: none"> - Issue one instruction per (fast) cycle - ALU takes multiple cycles 	Issue rate, FU stalls, FU depth
° Super-scalar	<ul style="list-style-type: none"> - Issue multiple scalar instructions per cycle 	Clock skew, FU stalls, FU depth
° VLIW ("EPIC")	<ul style="list-style-type: none"> - Each instruction specifies multiple scalar operations - Compiler determines parallelism 	Hazard resolution
° Vector operations	<ul style="list-style-type: none"> - Each instruction specifies series of identical operations 	Packing
		AVX extension, GPUs

77

Static multiple issue and dynamic multiple issue

1. Packaging instructions into **issue slots**: how does the processor determine how many instructions and which instructions can be issued in a given clock cycle?

In most static issue processors, this process is at least partially handled by the **compiler**; in dynamic issue designs, it is normally dealt with at runtime by the **processor**, although the compiler will often have already tried to help improve the issue rate by placing the instructions in a beneficial order.

2. Dealing with data and control hazards: in static issue processors, the compiler handles some or all the consequences of data and control hazards statically.

In contrast, most dynamic issue processors attempt to alleviate at least some classes of hazards using **hardware techniques operating at execution time**.

78

A design used in some embedded processors

Two instructions per cycle

Issuing two instructions per cycle will require fetching and decoding **64 bits of instructions**.

In many static multiple-issue processors, and essentially all VLIW (Very Long Instruction Word) processors, the layout of simultaneously issuing instructions is restricted to simplify the decoding and instruction issue.

We will require that the instructions be paired and aligned on a 64-bit boundary, with the ALU or branch portion appearing first. If one instruction of the pair cannot be used, we require that it be replaced with a **nop**.

Static multiple-issue processors vary in how they deal with potential data and control hazards.

In some designs, the **compiler** takes full responsibility for removing *all* hazards, scheduling the code, and inserting no-ops so that the code executes without any need for hazard detection or hardware-generated stalls.

In others, the **hardware** detects data hazards and generates stalls between two issue packets, while requiring that the **compiler** avoid all dependences within an instruction packet.

A hazard generally forces the entire issue packet containing the dependent instruction to stall.

Whether the software must handle all hazards or **only try to reduce the fraction of hazards** between separate issue packets, the appearance of having a large single instruction with multiple operations is reinforced.

79

Static two-issue pipeline in operation

Remove structural hazards:

- Two instructions in the same clock cycle (64-bit VLIW instruction).
- In one clock cycle, we may need to **read two registers** for the ALU operation and **two more for a store**, and also **one write** port for an ALU operation and **one write** port for a load.
 - 4 read operations and 2 write operations
- Since the ALU is tied up for the ALU operation, we also need a separate adder to calculate the effective address for data transfers. Without these extra resources, our two-issue pipeline would be hindered by structural hazards.

Clearly, this two-issue processor can improve performance by up to a factor of two!

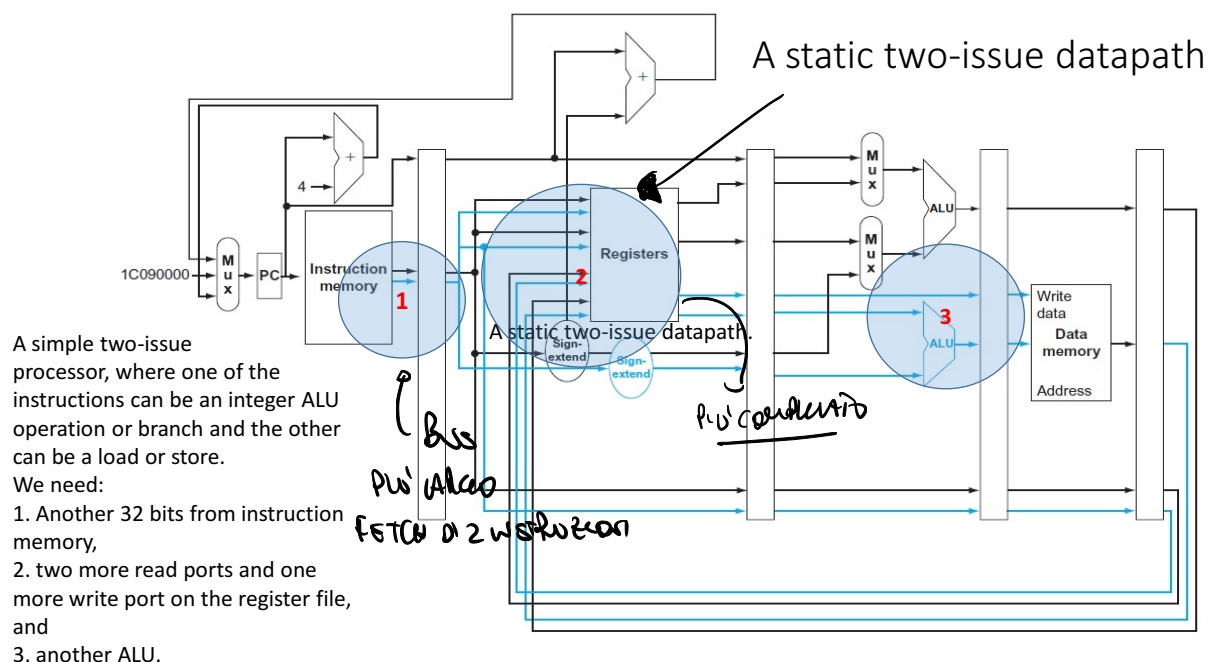
But, the relative performance loss from data and control hazards increases .

- For example, in the simple five-stage pipeline, loads have a **use latency** of one clock cycle, which prevents one instruction from using the result without stalling.
- In the two-issue, five-stage pipeline the result of a load instruction cannot be used on the next *clock cycle*. This means that the next two instructions cannot use the load result without stalling. Furthermore, ALU instructions that had no use latency in the simple five-stage pipeline now have a one-instruction use latency, since the results cannot be used in the paired load or store.

To effectively exploit the parallelism available in a multiple-issue processor, more ambitious compiler or hardware scheduling techniques are needed, and static multiple issue requires that the compiler take on this role.

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction					IF	ID	EX	MEM
ALU or branch instruction						IF	ID	EX
Load or store instruction							IF	ID

80



81

Very Long Instruction Word (VLIW) processors

The compiler role:

- The compiler's responsibilities may include **static branch prediction** and **code scheduling** to reduce or prevent all hazards.

How can the designer achieve the scalability of this type of microprocessor?

- ???

What may be the next evolution steps?

- ???

82

Core Processor overloading situation?

SE AGREGANDO PROCESSING UNIT POR CADA INSTRUÇÃO PARA CADA CORE POSSO ESCALAR
PARALELAMENTE!

Superscalar processors, Dynamic multiple-issue processors

- In the simplest superscalar processors, instructions issue in order, and the processor decides whether zero, one, or more instructions can issue in a given clock cycle.
- Achieving good **performance** on such a processor still requires the **compiler** to try to schedule instructions to move dependences apart.
- Even with such compiler scheduling, there is an important difference between this simple superscalar and a VLIW processor: **the code**, whether scheduled or not, **is guaranteed by the hardware to execute correctly**.
- Compiled code will **always run correctly independent of the issue rate or pipeline structure** of the processor.
- In some **VLIW designs**, this has not been the case, and **recompilation** was required when moving across different processor models.

83

Se il SW è strutturato in modo sequenziale non ha vantaggi

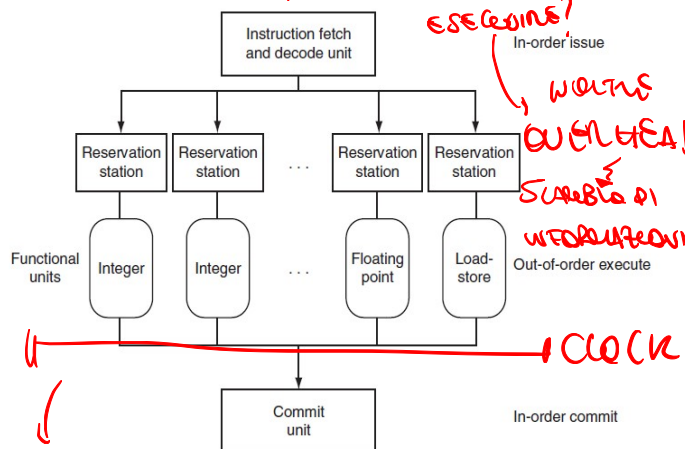
Dynamic Pipeline Scheduling (I) superscalar out-of-order

Dynamic pipeline scheduling chooses which instructions to execute next, possibly reordering them to avoid stalls.

The pipeline is divided into three major units: an instruction fetch and issue unit, multiple functional units, and a **commit unit**.

This is also a model for the software architecture.

The sequence of instructions performed **asynchronously and in parallel** in their heaviest part from the point of view of computing power.



Il codice è dato dal SW;

Quante istruzioni in parallelo posso dare allo stesso tempo?

In-order issue

Waiting OVERHEAD!

Scambio di informazioni

Out-of-order execute

In-order commit

Order Code

Value ESSESSO di istruzioni

84

RECODE:

1) non viene processata l'istruzione

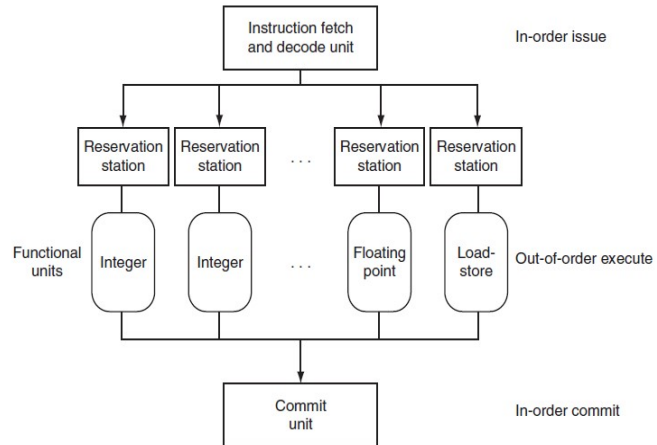
2) viene riammessa l'istruzione

Overcoming Data Hazards With Dynamic Scheduling

The hardware reorders the instruction execution to reduce the stalls while maintaining data flow and exception behavior.

Dynamic scheduling offers several advantages.

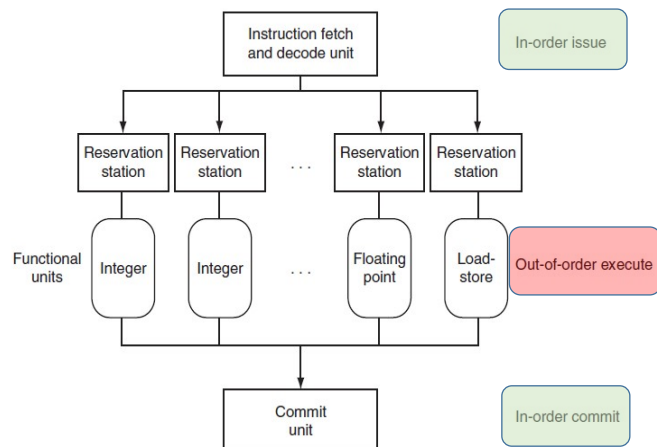
- **First**, it allows code that was compiled with one pipeline in mind **to run efficiently on a different pipeline**, eliminating the need to have multiple binaries and recompile for a different microarchitecture.
 - Software is from third parties and distributed in binary form.
- **Second**, it enables handling some cases when dependences are unknown at compile time.
 - For example, they may involve a memory reference or a data dependent branch, or they may result from a modern programming environment that uses dynamic linking or dispatching.
- **Third**, it allows the processor to tolerate unpredictable delays, such as cache misses, by executing other code while waiting for the miss to resolve.



85

Superscalar out-of-order (1)

- The first unit fetches instructions, decodes them, and sends each instruction to a corresponding functional unit for execution.
- Each functional unit has buffers, called **reservation stations**, which hold the operands and the operation.
- As soon as the buffer contains all its operands and the functional unit is ready to execute, the result is calculated.
- When the result is completed, it is sent to any reservation stations waiting for this particular result (forwarding) as well as to the commit unit, which buffers the result until it is safe to put the result into the register file or, for a store, into memory.
- The buffer in the commit unit, often called the **reorder buffer**, is also used to supply operands, in much the same way as forwarding logic does in a statically scheduled pipeline.
- Once a result is committed to the register file, it can be fetched directly from there, just as in a normal pipeline.



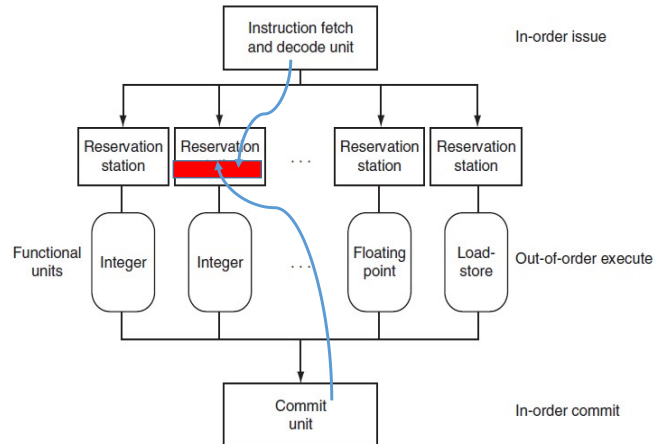
86

Superscalar out-of-order (I)

The combination of buffering operands in the reservation stations and results in the reorder buffer provides a form of register renaming.

To see how this conceptually works, consider the following steps:

1. When an instruction issues, it is copied to a reservation station for the appropriate functional unit.
 - Any operands that are available in the register file or reorder buffer are also immediately copied into the reservation station.
 - The instruction is buffered in the reservation station until all the operands and the functional unit are available.
 - For the issuing instruction, the register copy of the operand is no longer required, and if a write to that register occurred, the value could be overwritten.

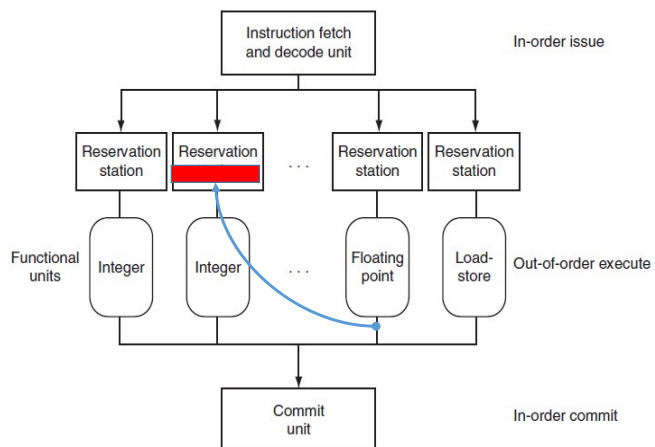


87

Superscalar out-of-order (II)

2. If an operand is not in the register file or reorder buffer, it must be waiting to be produced by a functional unit.

- The name of the functional unit that will produce the result is tracked.
- When that unit eventually produces the result, it is copied directly into the waiting reservation station from the functional unit **bypassing** the registers.



88

Register renaming (1)

- Original code

ist1: op **R1**,R2,R3

ist2: op R4,R5,**R1**

- Modified code

ist1: op **R1**,R2,R3

ist2: op R4,R5,**R1***

R1*  **R1** for each destination register

89

Register renaming (2)

Original code

ist1: op R1,R2,**R3**

- instructions using R3

ist2: op R4,R5,**R3**

- instructions using R3



Modified code

ist1: op R1,R2,**R3**

- instructions using R3

ist2: op R4,R5,**R3***

- instructions using R3*


R3*  **R3**

90

Register renaming (3)

all the register are renamed

Original

1 - op R1,R2,**R3**

2 - op R3,R2,R4

3 - op R4,R3,R5

4 - op R2,R4,**R3**

5 - op R3,R2,R6

Architectural register renamed
at execution time

1 - op R1*,R2*,**R3***

2 - op R3*,R2*,R4*

3 - op R4*,R3*,R5*

4 - op R2*,R4*,**R3****

5 - op R3**,R2*,R6*

64 architectural registers: R0, R1, R2,, R63

1K physical registers: PR0,, PR1023

91

Register renaming (4)

Allocate and deallocate physical registers

Original

1 - op R1,R2,**R3**

2 - op **R3**,R2,R4

3 - op R4,**R3**,R5

.....

4 - op R2,R4,**R3**

5 - op **R3**,R2,R6

Architectural register renamed
at run time

1 - op R1*,R2*,**R3*** rename R1, R2 **and allocate new physical register R3***

2 - op **R3***,R2*,R4* rename R3, R2 and allocate new physical register R4*

3 - op R4*,**R3***,R5* rename R4, R3 and allocate new physical register R5*

4 - op R2*,R4*,**R3**** rename R2, R4 **allocate a new physical register R3***

5 - op **R3****,R2*,R6* rename R3, R2 and allocate new physical register R6*

92

The out-of-order execution (1)

Arch. reg.	Original Instr.	Renamed Instr.	Physical/arch. ptr.	
R1= 4	1) op R1, R2, R3	1) op R2*, R4*, R1*	R1*=	3
R2= 3	2) op R3,R2,R4		R2*= 4	1
R3= 7	3) op R1,R5,R3		R3*=	
R4= 1	4) op R3,R1,R5		R4*= 3	2
R5= 9			R5*=	
R6= 5			R6*=	
			R7*=	
			R8*=	
			R9*=	
			R10*=	

93

The out-of-order execution (2)

Arch. reg.	Original Instr.	Renamed Instr.	Physical/arch. ptr.	
R1= 4	1) op R1, R2, R3	1) op R2*, R4*, R1*	R1*=	3
R2= 3	2) op R3,R2,R4		R2*= 4	1
R3= 7	3) op R1,R5,R3	3) op R2*,R7*,R5*	R3*=	
R4= 1	4) op R3,R1,R5		R4*= 3	2
R5= 9			R5*=	3
R6= 5			R6*=	
			R7*= 9	5
			R8*=	
			R9*=	
			R10*=	

94

The out-of-order execution (3)

Arch. reg.	Original Instr.	Renamed Instr.	Physical/arch. ptr.
R1= 4	1) op R1, R2, R3	1) op R2*, R4*, R1*	R1*= 3
R2= 3	2) op R3,R2,R4	3) op R2*,R7*,R5*	R2*= 4
R3= 7	3) op R1,R5,R3	4) op R4*,R2*,R6*	R3*=
R4= 1	4) op R2,R1,R5		R4*= 3
R5= 9			R5*= 3
R6= 5			R6*= 5
			R7*= 9
			R8*=
			R9*=
			R10*=

95

The out-of-order execution (4)

Arch. reg.	Original Instr.	Renamed Instr.	Physical/arch. ptr.
R1= 4	1) op R1, R2, R3	1) op R2*, R4*, R1*	R1*= 2
R2= 3	2) op R3,R2,R4	3) op R2*,R7*,R5*	R2*= 4
R3= 2	3) op R1,R5,R3	4) op R4*,R2*,R6*	R3*= 4
R4= 1	4) op R2,R1,R5	2) op R1*,R4*,R3*	R4*= 3
R5= 9			R5*= 3
R6= 5			R6*= 5
			R7*= 9
			R8*=
			R9*=
			R10*=

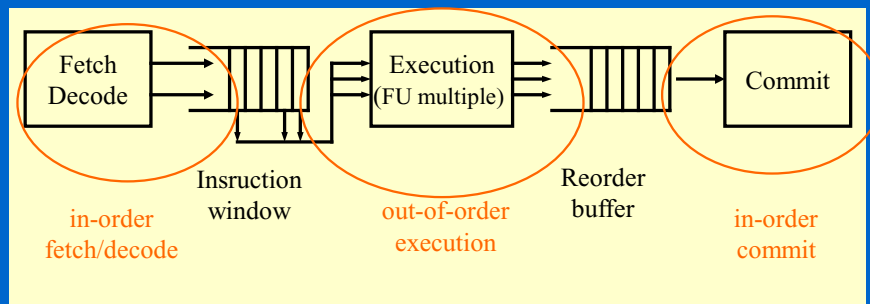
96

The out-of-order execution (5)

Arch. reg.	Original Instr.	Renamed Instr.	Physical/arch. ptr.
R1= 4	1) op R1, R2, R3	1) op R2*, R4*, R1*	R1*=2 3
R2= 3	2) op R3,R2,R4	3) op R2*,R7*,R5*	R2*= 4 1
R3= 2	2b) op R3,R3,R6	4) op R4*,R2*,R6*	R3*= 4
R4= 1 <i>Using R3</i>	2) op R1*,R4*,R3*	R4*= 3 2
R5= 9	3) op R1,R5,R3	2b) op R1*,R1*,R8*	R5*=10 3
R6= 5	4) op R2,R1,R5 <i>Using R1*</i>	R6*= 5 5
			R7*= 9 5
			R8*= 6
			R9*=
			R10*=

97

Commit



- The pipeline is split in 3 zones:
 - in-order: fetch and decode
 - out-of-order: execution
 - in-order: commit

98

speculative execution

Es.: *original code*

```

if ( x > 3 ){
    instr. 1
    instr. 2
}
else
    instr. 3
instr. 4

```

modified code

```

pred = ( x>3 )
instr.1 (pred - true)
instr.2 (pred - true)
instr.3 (pred - false)
instr.4

```

Oss.: no conditional jumps

When the predicate value is ready (compare x to 3), the commit is performed considering the predicate value for each statement.

The value is used to decide whether to commit (write back the result) or delete the result.

99

speculative execution

original code

```

if ( x > 3 ){
    instr. 1
    instr. 2
    if ( y > 5 ){
        instr. 5
        instr. 6
    }
}
else
    instr. 3
instr. 4

```

modified code

```

Rpred1 = (x>3)
instr.1 (Rpred1 - true)
instr.2 (Rpred1 - true)
Rpred2 = (y>5) (Rpred1 - true)
instr.5 (Rpred2 - true)
instr.6 (Rpred2 - true)

instr.3 (Rpred1 - false)
instr. 4 (true)

```

100

The technologies used on the past

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue Width	Out-of-Order/Speculation	Cores/Chip	Power	
Intel 486	1989	25 MHz	5	1	No	1	5	W
Intel Pentium	1993	66 MHz	5	2	No	1	10	W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29	W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75	W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103	W
Intel Core	2006	2930 MHz	14	4	Yes	2	75	W
Intel Core i5 Nehalem	2010	3300 MHz	14	4	Yes	2–4	87	W
Intel Core i5 Ivy Bridge	2012	3400 MHz	14	4	Yes	8	77	W

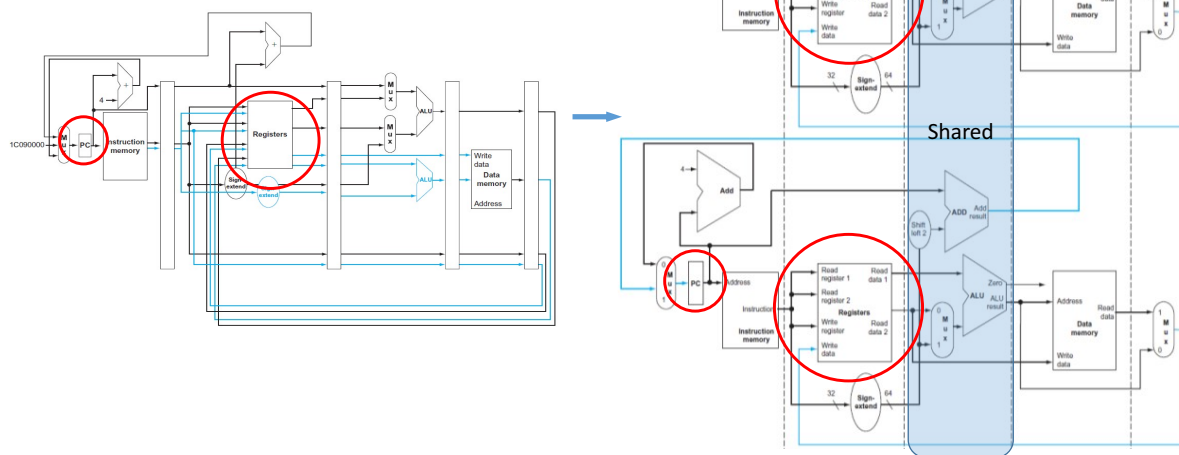
101

Multithreading

Multithreading allows multiple threads to share the functional units of a single processor in an overlapping fashion.

102

Two different ways
to put two pipelines together



103

Hardware Multithreading

A **thread** is like a process in that it has state and a current program counter, but threads typically share the address space of a single process, allowing a thread to easily access data of other threads within the same process.

Hardware multithreading allows multiple threads to share the functional units of a *single* processor in an overlapping fashion to try to utilize the hardware resources efficiently.

Multithreading, however, does not duplicate the entire processor as a multiprocessor does.

- To permit this sharing, the processor must duplicate the independent state of each thread.
 - For example, each thread would have a separate copy of the **register file** and the **program counter**.
- The memory itself can be shared through the virtual memory mechanisms, which already support multi-programming.
- The hardware must support the ability to change to a different thread relatively quickly.
 - A thread switch should be much more efficient than a process switch, which typically requires hundreds to thousands of processor cycles while a thread switch can be instantaneous.

For multithreading hardware to achieve performance improvements, a program must contain multiple threads (we sometimes say that the application is multithreaded) that could execute in concurrent fashion. These threads are identified either by a compiler (typically from a language with parallelism constructs) or by the programmer.

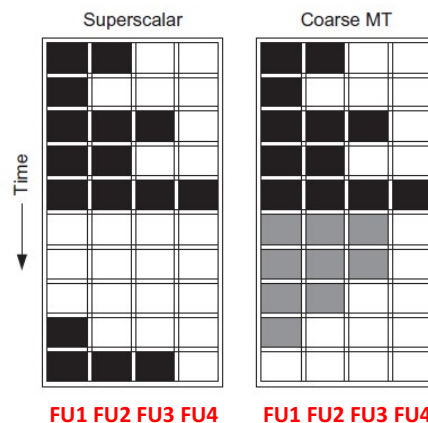
Many recent processors incorporate both multiple processor cores on a single chip and provide multithreading within each core.

104

Hardware Multithreading

Coarse-grained multithreading

- The processor executes instructions from a single thread, when a stall occurs, the pipeline will see a bubble before the new thread begins executing.
- Coarse-grained multithreading **switches** threads only on costly stalls, such as **level two or three cache misses**.
- Coarse-grained multithreading relieves the need to have **thread-switching** be essentially free and is much less likely to **slow down the execution of any one thread**.
- The main limitation of this model arises from the pipeline **start-up costs** of coarse-grained multithreading.
- Because of this start-up overhead, coarse-grained multithreading is much more useful for reducing the penalty of very high-cost stalls, where pipeline refill is negligible compared to the stall time.

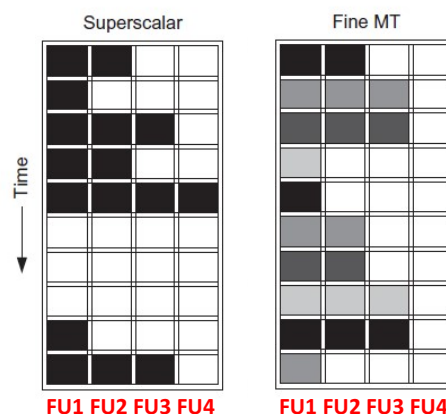


105

Hardware Multithreading

Fine-grained multithreading

- **Fine-grained multithreading** switches between threads on each instruction, resulting in interleaved execution of multiple threads.
- This interleaving is often done in a round-robin fashion, skipping any threads that are stalled at that clock cycle.
- To make fine-grained multithreading practical, the processor must be able to switch threads on every clock cycle.
- **One advantage** of fine-grained multithreading is that it can hide the throughput losses that arise from both short and long stalls, since instructions from other threads can be executed when one thread stalls.
- **The primary disadvantage** of fine-grained multithreading is that it slows down the execution of the individual threads, since a thread that is ready to execute without stalls will be delayed by instructions from other threads.



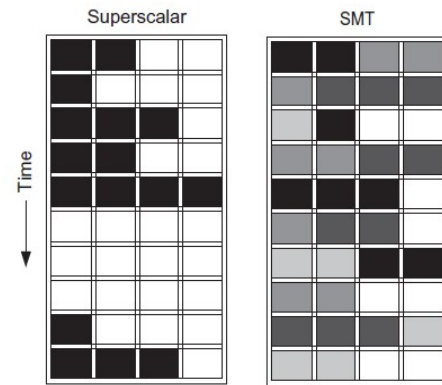
106

Simultaneous multithreading (SMT)

Simultaneous multithreading is a variation on fine-grained multithreading that arises naturally when fine-grained multithreading is implemented on top of a multiple-issue, dynamically scheduled processor.

SMT uses thread-level parallelism to hide long-latency events in a processor, thereby increasing the usage of the functional units.

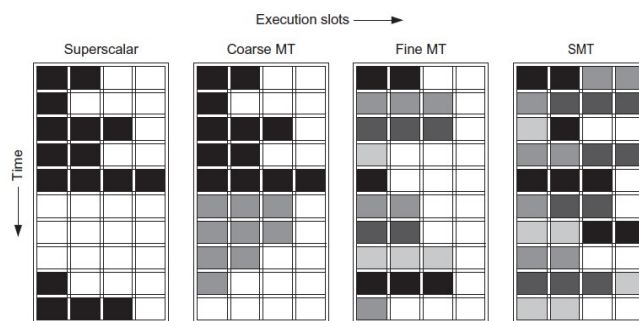
The key insight in SMT is that register renaming and dynamic scheduling allow multiple instructions from independent threads to be executed without regard to the dependences among them; the resolution of the dependences can be handled by the dynamic scheduling capability.



107

multithreaded processors

- The Sun T1 and T2 (aka Niagara) processors are fine-grained, multithreaded processors, while the Intel Core i7 and IBM Power7 processors use SMT.
- The T2 has 8 threads, the Power7 has 4, and the Intel i7 has 2.



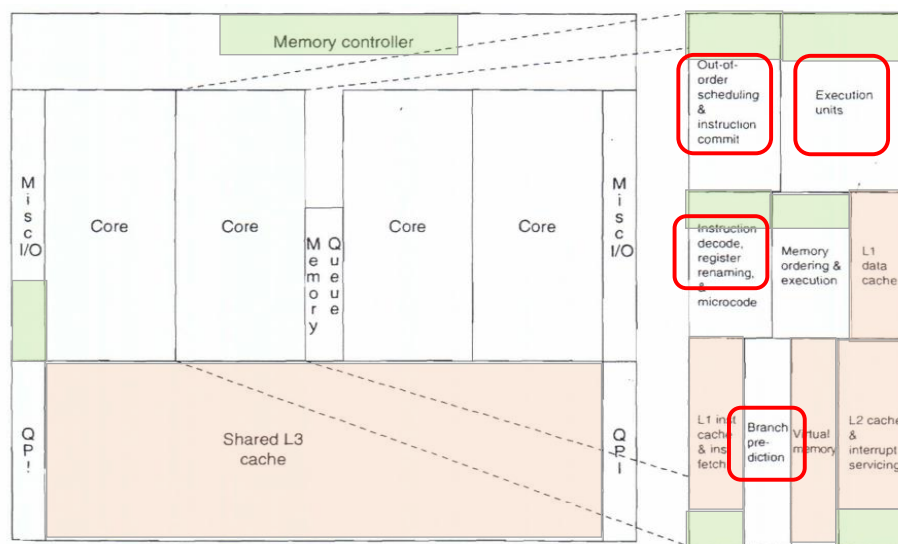
108

Multithreading out-of-order processor

- Simultaneous multithreading uses the insight that a dynamically scheduled processor already has many of the hardware mechanisms needed to support the mechanism, including a large virtual register set.
- Multithreading can be built on top of an out-of-order processor by adding a **per-thread renaming table**, keeping separate PCs, and providing the capability for instructions from multiple threads to commit.

109

Intel core i7



110

110

The Intel Core i7

The first i7 processor was introduced in 2008.

The i7 uses an aggressive out-of-order speculative microarchitecture with deep pipelines with the goal of achieving high instruction throughput by combining multiple issue and high clock rates.

111

The Intel Core i7

The big problem derives from wanting to maintain the compatibility of the instruction set with the architectures of the past.

The machine language level architecture is CISC:

- variable length of the instructions
- complex instruction set (complex semantics)
- variable execution time
- complex ways of addressing memory
- numerous accesses to memory required (operands in memory)

while the micro-code executed is RISC.

This conditions the architecture and, the fetch and decoding of the instructions.

112

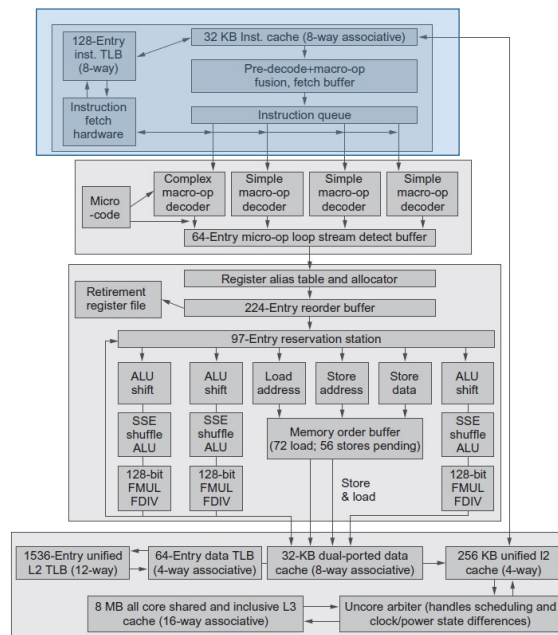
1. Instruction fetch

The processor uses a sophisticated multilevel branch predictor to achieve a balance between speed and prediction accuracy.

There is also a return address stack to speed up function return.

Mispredictions cause a penalty of about 17 cycles.

Using the predicted address, the instruction fetch unit fetches 16 bytes from the instruction cache.



113

2. The predecode instruction buffer and the instruction queue

The 16 bytes are placed in the predecode instruction buffer.

Macro-op fusion takes instruction combinations such as compare followed by a branch and fuses them into a single operation, which can issue and dispatch as one instruction.

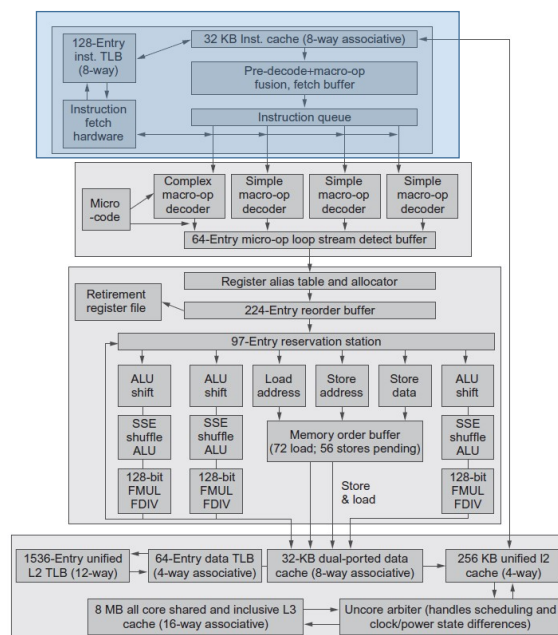
Only certain special cases can be fused, since we must know that the only use of the first result is by the second instruction (i.e., compare and branch).

The macrofusion has a significant impact on the performance of integer programs resulting in an 8%–10% average increase in performance with a few programs showing negative results. Little impact on FP programs.

The predecode stage also breaks the 16 bytes into individual x86 instructions.

The length of an x86 instruction can be from 1 to 17 bytes and the predecoder must look through a number of bytes before it knows the instruction length.

Individual x86 instructions (including some fused instructions) are placed into the instruction queue.



114

3. Micro-op decode

—Individual x86 instructions are translated into micro-ops.

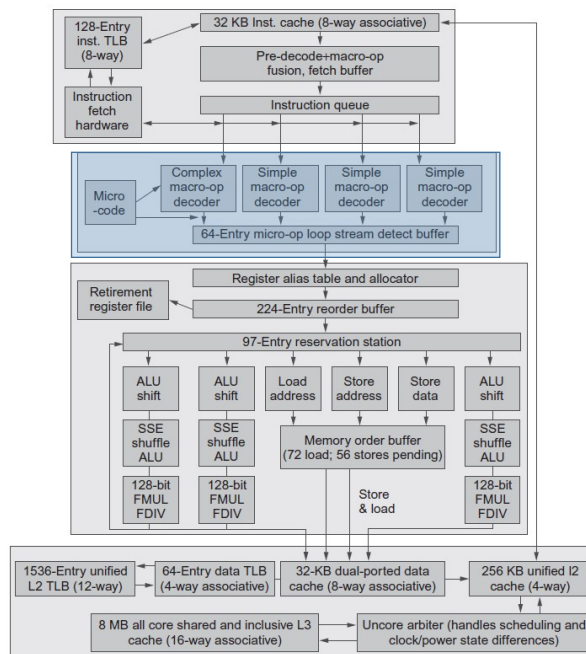
Micro-ops are simple RISC-V-like instructions that can be executed directly by the pipeline. Introduced in the Pentium Pro in 1997.

Three of the decoders handle x86 instructions that translate directly into one micro-op.

For x86 instructions with more complex semantics, there is a microcode engine that is used to produce the micro-op sequence.

It can produce up to four micro-ops every cycle and continues until the necessary micro-op sequence has been generated.

The microops are placed according to the order of the x86 instructions in the 64-entry micro-op buffer.



115

4. The loop stream detection and microfusion

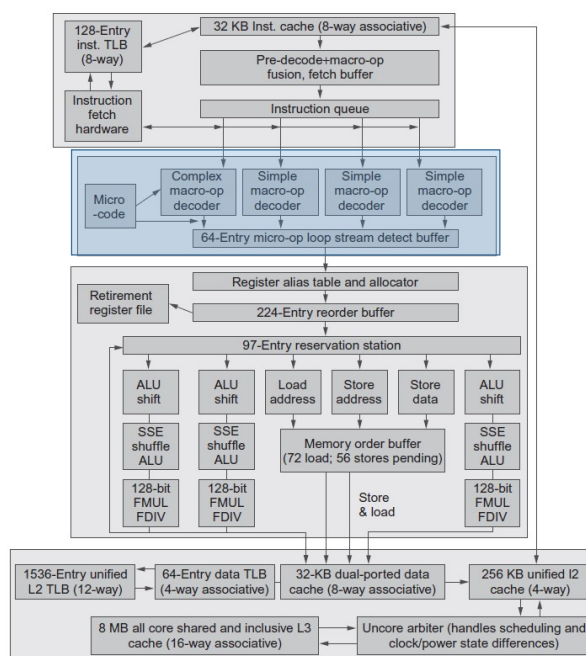
If there is a small sequence of instructions (less than 64 instructions) that comprises a loop, the loop stream detector

- will find the loop and
- directly issue the micro-ops from the buffer.

eliminating the need for the instruction fetch and instruction decode stages to be activated.

Microfusion combines instruction pairs such as ALU operation and a dependent store and issues them to a single reservation station (where they can still issue independently), thus increasing the usage of the buffer.

Micro-op fusion produces smaller gains for integer programs and larger ones for FP.



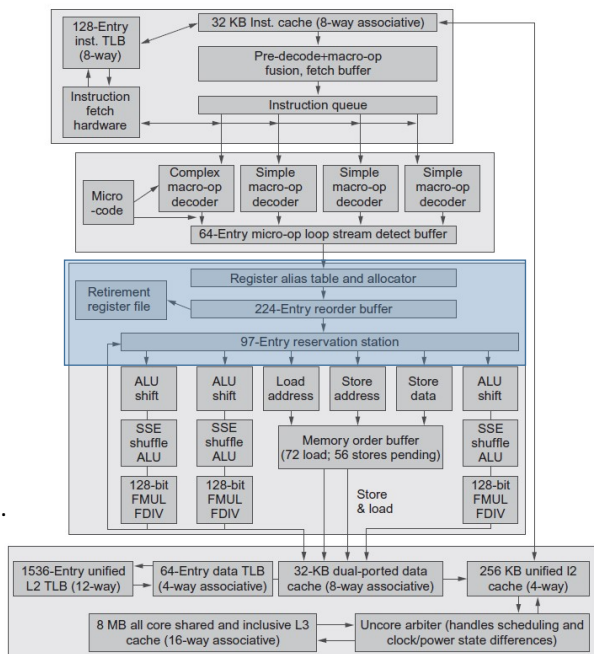
116

5. Perform the basic instruction issue

- Looking up the register location in the register tables, renaming the registers,
- allocating a reorder buffer entry,
- and fetching any results from the registers or reorder buffer

before sending the micro-ops to the reservation stations.

Up to four micro-ops can be processed every clock cycle; they are assigned the next available reorder buffer entries.

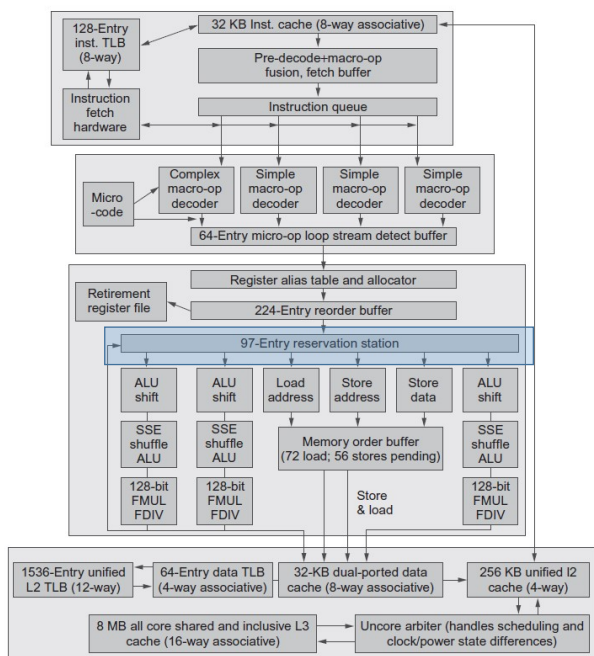


117

6. The centralized reservation station

The i7 uses a centralized reservation station shared by six functional units.

Up to six micro-ops may be dispatched to the functional units every clock cycle.



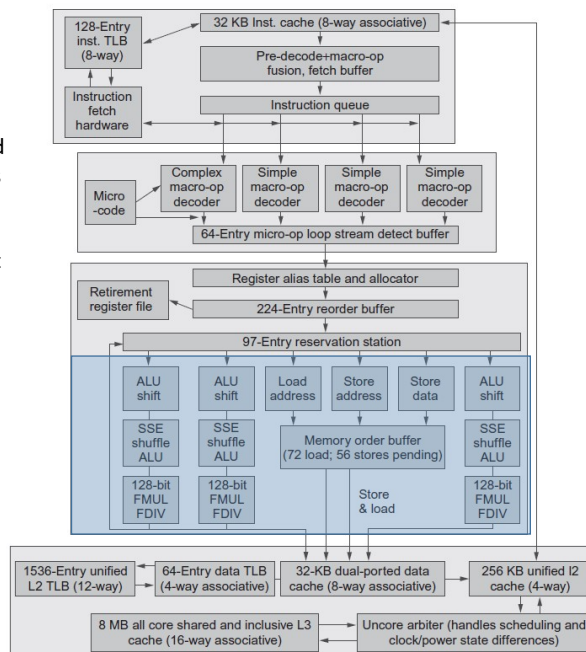
118

7. Micro-ops execution

7. Micro-ops are executed by the individual function units, and then results are sent back to any waiting reservation station as well as to the register retirement unit.

Where they will update the register state once it is known that the instruction is no longer speculative.

The entry corresponding to the instruction in the reorder buffer is marked as complete.

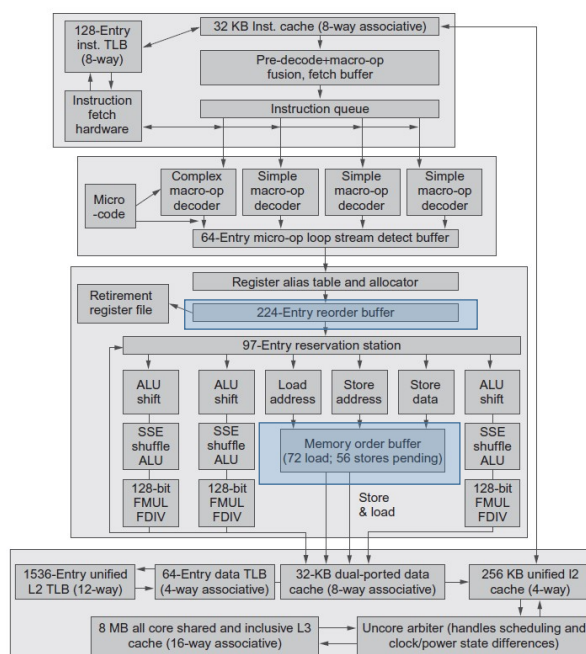


119

8. Commit

When one or more instructions at the head of the reorder buffer have been marked as complete,

- the pending writes in the register retirement unit are executed, and
- the instructions are removed from the reorder buffer.



120

The ARM Cortex-A53 pipeline

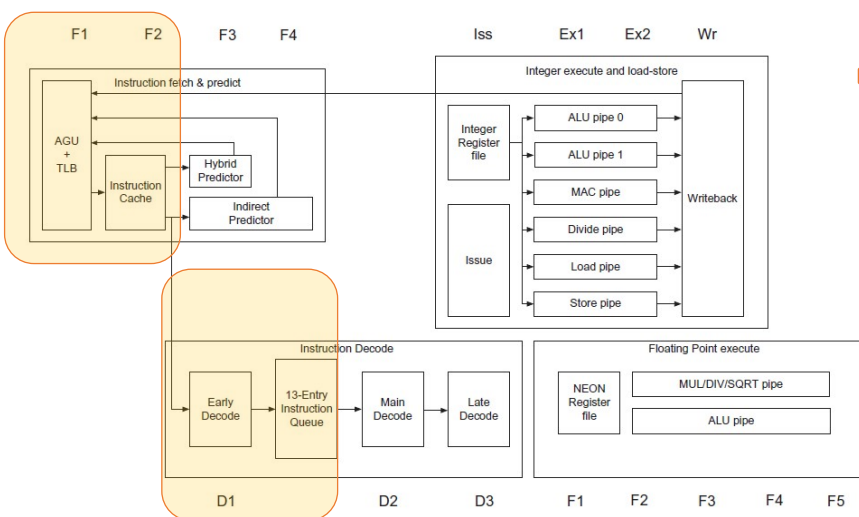
121

Specification of the ARM Cortex-A53 and the Intel Core i7 920

Processor	ARM A53	Intel Core i7 920
Market	Personal Mobile Device	Server, Cloud
Thermal design power	100 milliWatts (1 core @ 1 GHz)	130 Watts
Clock rate	1.5 GHz	2.66 GHz
Cores/Chip	4 (configurable)	4
Floating point?	Yes	Yes
Multiple Issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline Stages	8	14
Pipeline schedule	Static In-order	Dynamic Out-of-order with Speculation
Branch prediction	Hybrid	2-level
1st level caches/core	16-64 KiB I, 16-64 KiB D	32 KiB I, 32 KiB D
2nd level cache/core	128–2048 KiB (shared)	256 KiB (per core)
3rd level cache (shared)	(platform dependent)	2–8 MiB

122

The ARM Cortex-A53



The Address Generation Unit (AGU) uses a Hybrid Predictor, Indirect Predictor, and a Return Stack to predict branches to try to keep the instruction queue full.

- The first three stages fetch instructions into a 13-entry instruction queue.

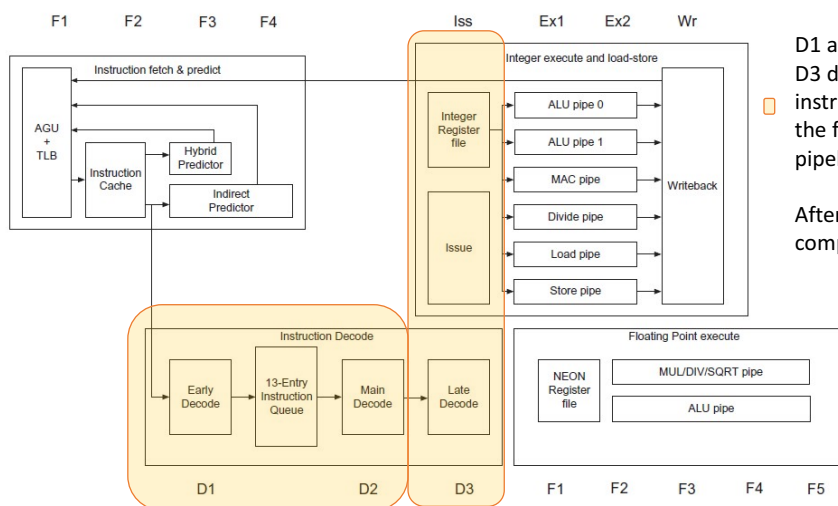
The four cycles of instruction fetch include an address generation unit that produces the next PC either by incrementing the last PC or from one of three predictors.

The target cache is checked during the first fetch cycle, if it hits; then the next two instructions are supplied from the target cache.

In case of a hit and a correct prediction, the branch is executed with no delay cycles.

123

The ARM Cortex-A53

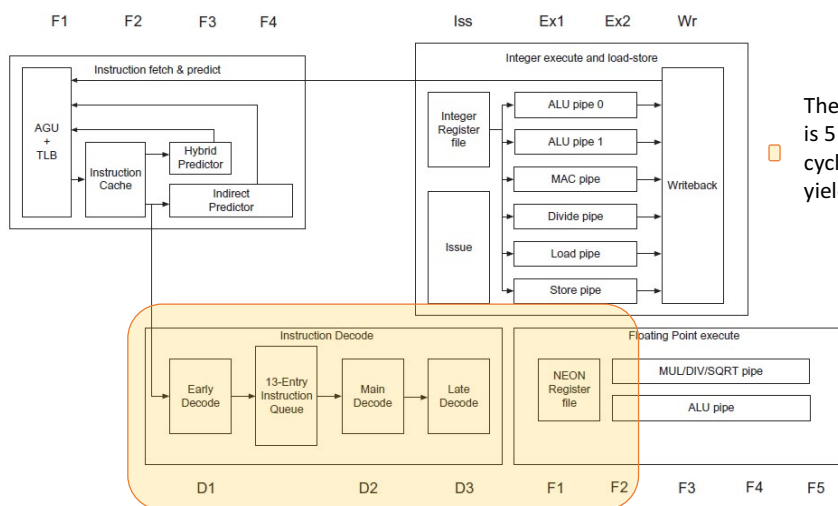


- D1 and D2 do the basic decoding, and D3 decodes some more complex instructions and is overlapped with the first stage of the execution pipeline (ISS).

After ISS, the Ex1, Ex2, and WB stages complete the integer pipeline.

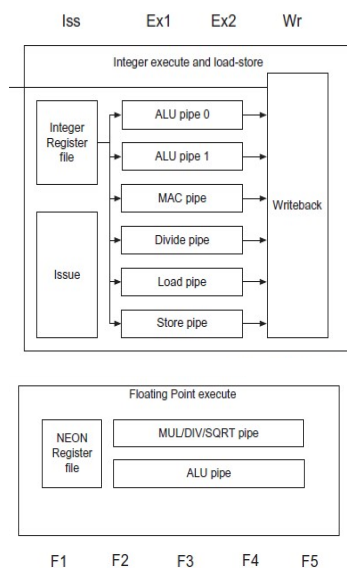
124

The ARM Cortex-A53



125

ARM Cortex-A53



126

The **decode stages** of the pipeline **determine if there are dependencies** between a pair of instructions, which would **force sequential execution**, and in which pipeline of the execution stages to send the instructions.

The instruction execution section primarily occupies three pipeline stages and provides **one pipeline for load instructions, one pipeline for store instructions, two pipelines for integer arithmetic operations, and separate pipelines for integer multiply and divide operations**. Either instruction from the pair can be issued to the load or store pipelines.

The execution stages have **full forwarding between the pipelines**.

Floating-point and SIMD operations add a two more pipeline stages to the instruction execution section and feature one pipeline for multiply/divide/square root operations and one pipeline for other arithmetic operations.

Cortex-A53 is a configurable core

delivered as an IP (Intellectual Property) core

IP cores are the dominant form of technology delivery in the embedded, personal mobile device, and related markets; billions of ARM and MIPS processors have been created from these IP cores.

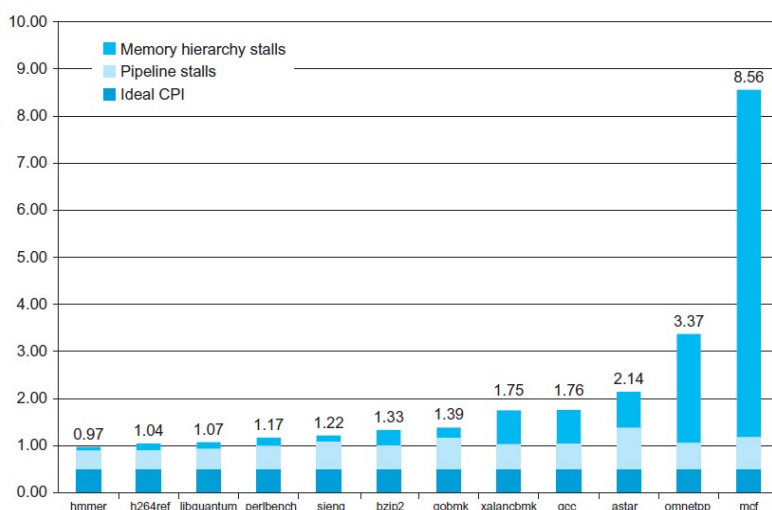
An IP core (which may itself be a multicore) is designed to be incorporated with other logic (hence it is the “core” of a chip), including application-specific processors (such as an encoder or decoder for video), I/O interfaces, and memory interfaces, and then fabricated to yield a processor optimized for a particular application.

Although the processor core is almost identical logically, the resultant chips have many differences.

One parameter is the size of the L2 cache, which can vary by a factor of 16.

127

CPI of the Cortex-A53 using the SPEC2006 benchmarks



While the ideal **CPI** is **0.5**, the best case achieved is 1.0, the median case is 1.3, and the worst case is 8.6.

For the median case, **60%** of the stalls are due to the **pipelining hazards** and **40%** are stalls due to the **memory hierarchy**.

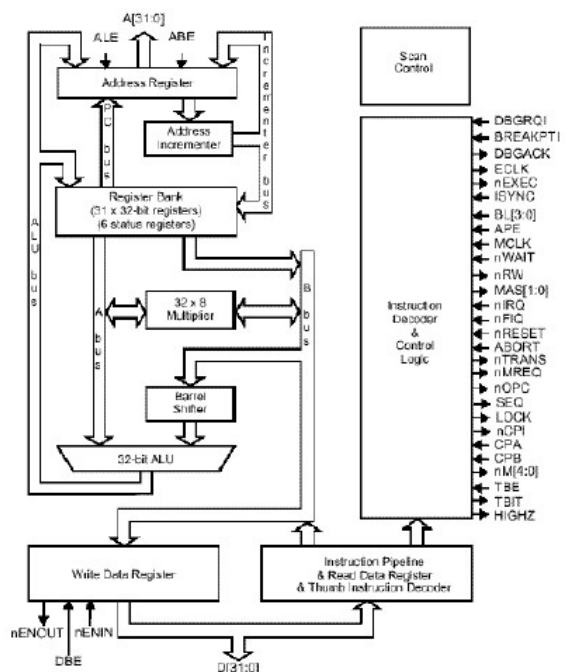
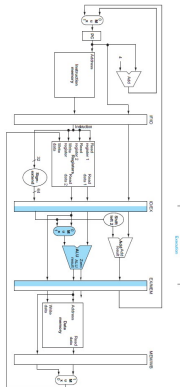
Pipeline stalls are caused by **branch mispredictions**, **structural hazards**, and **data dependencies** between pairs of instructions.

Given the **static pipeline in-order** of the Cortex-A53, it is up to the **compiler** to **try to avoid** structural hazards and data dependencies.

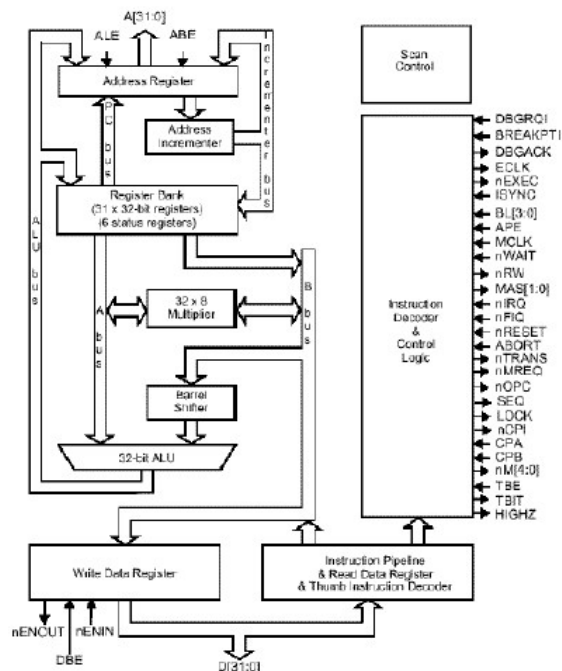
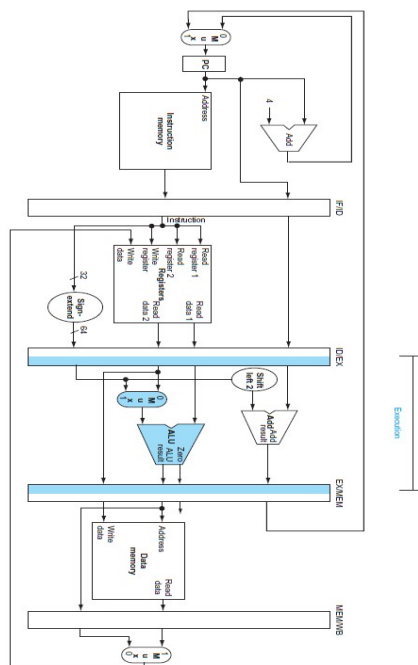
128

ARMv7TDMI

- Simple 3 stage pipeline
 - Fetch, Decode, Execute
 - Multiple cycles in execute stage for Loads/Stores
- Simple core
 - "Roll your own memory system"



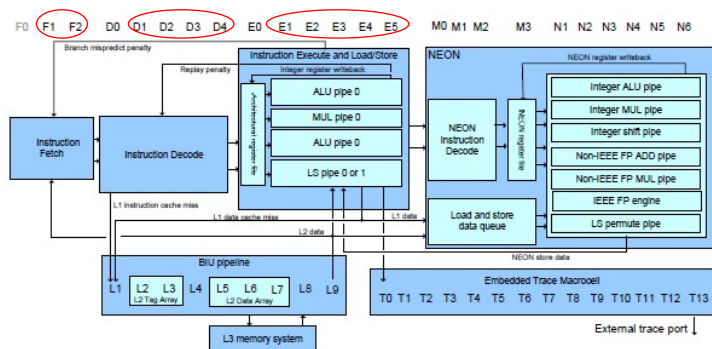
129



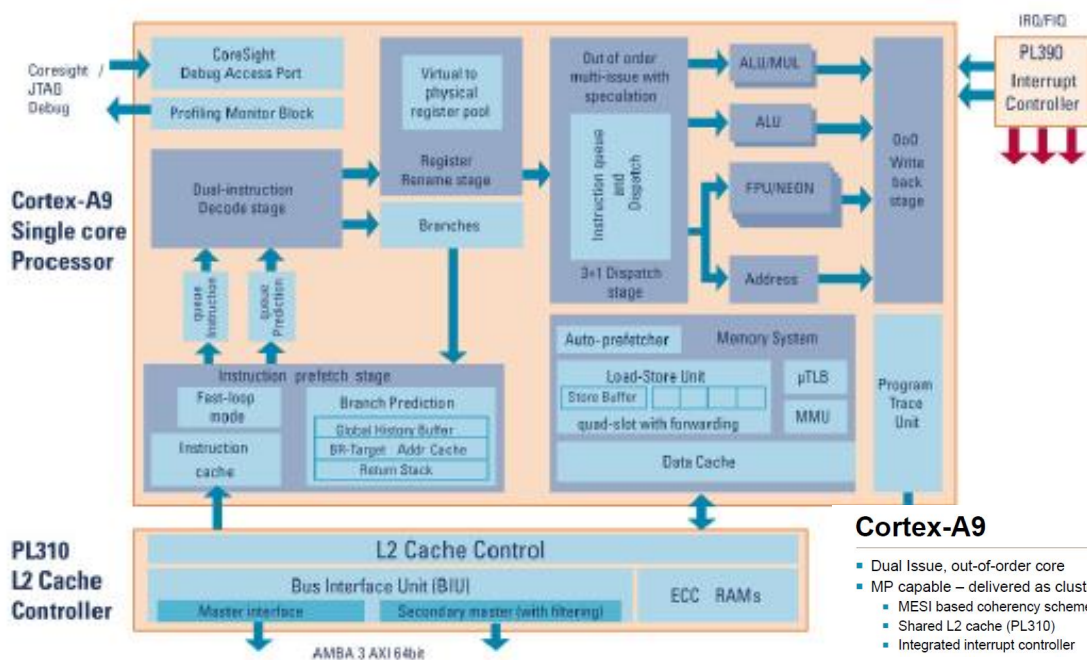
130

Cortex A8, 10 stage pipeline

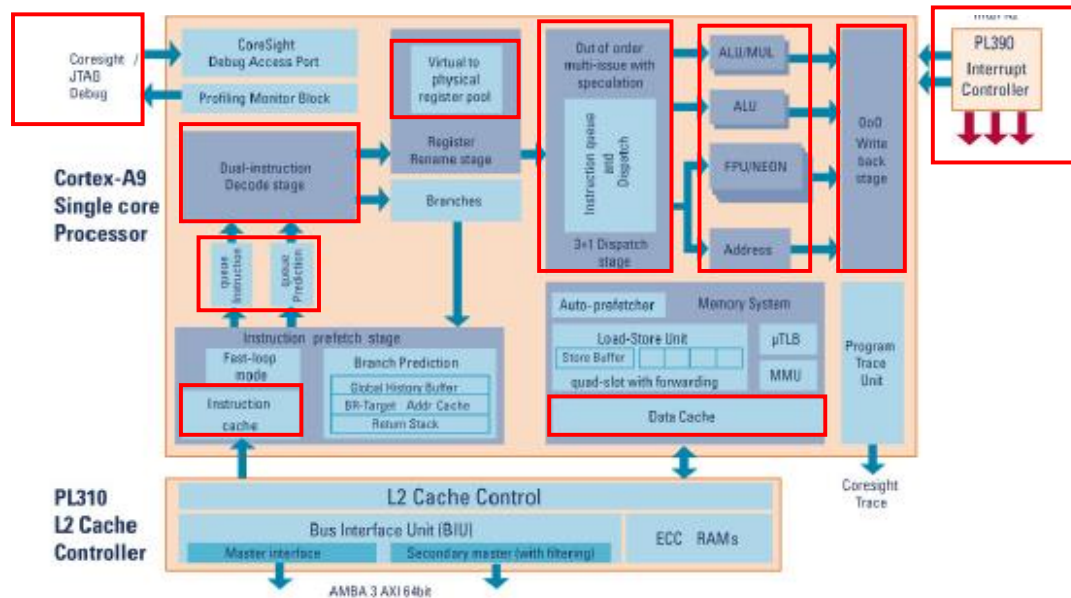
- Dual Issue, in-order
 - 10 stage pipeline (+ Neon Engine)
- 2 levels of cache – L1 I/D split, L2 unified
- Aggressive Branch Prediction



131



132



133

Power-Saving Instructions

Battery-powered devices save power by spending most of their time in sleep mode.

ARMv6K introduced instructions to support such power savings:

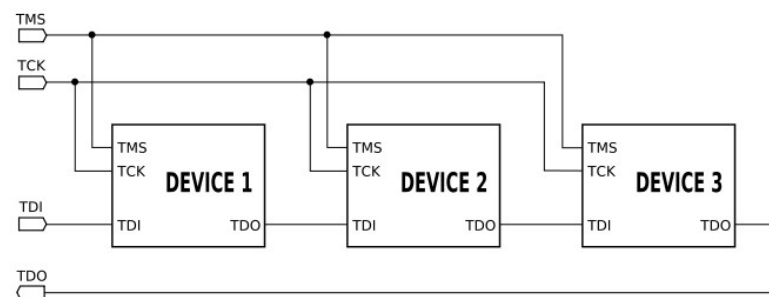
- The wait for interrupt (**WFI**) instruction allows the processor to enter a low-power state until an interrupt occurs.
- The system may generate interrupts based on user events (such as touching a screen) or on a periodic timer.
- The wait for event (**WFE**) instruction is similar but is helpful in multiprocessor systems so that a processor can go to sleep until notified by another processor.
- It wakes up either during an interrupt or when another processor sends an event using the **SEV** instruction.

134

JTAG

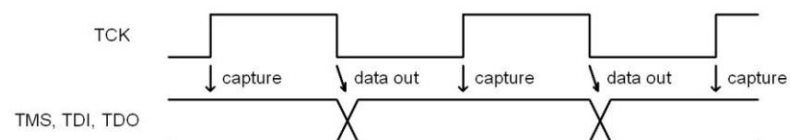
135

JTAG (IEEE 1149.1)



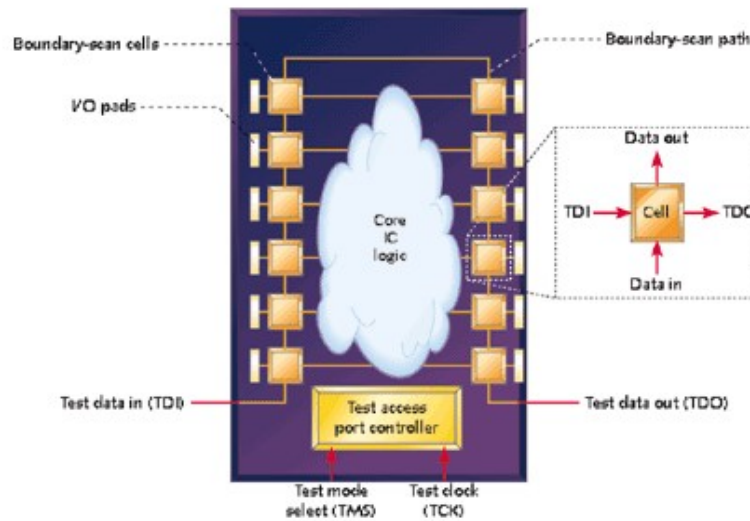
The connector pins are:

1. **TDI** (Test Data In)
2. **TDO** (Test Data Out)
3. **TCK** (Test Clock)



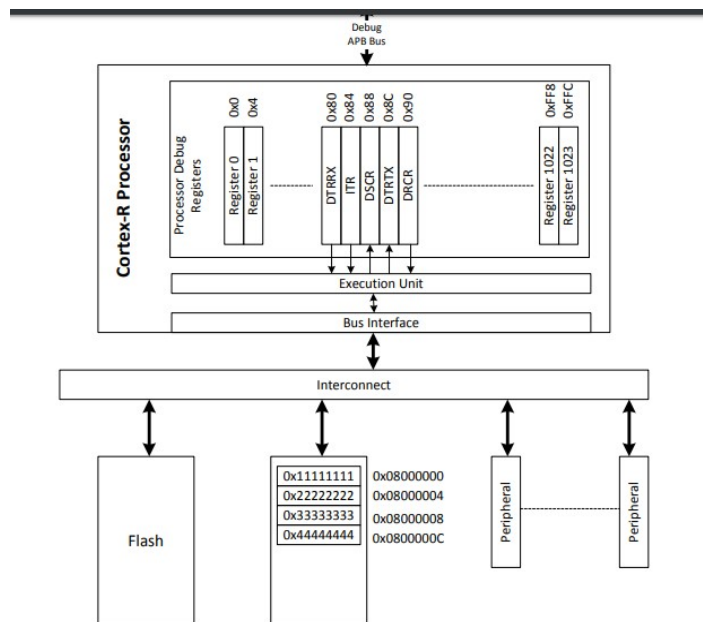
136

An integrated circuit with boundary scan



137

The Jtag scheme



138