Introducing modern computer architectures

- a simple RISC architecture
- single cycle instruction execution
- multi-cycle instruction execution
- RISC vs CISC architectures

Basic concepts

- **Microarchitecture**: the internal structure of a processor, and its **datapath**: namely, the path instructions go through when executed.

  In the following, we shall use freely the terms "microarchitecture", "architecture" or "internal architecture" (in a processor).

- **ISA**: Instruction Set Architecture. The set of machine instructions of a processor. Two distinct processors can have the same ISA but different microarchitectures (that is, two different ways to execute those instructions)

A simple RISC architecture

- We begin with a simple **RISC microarchitecture** (can you tell what RISC stands for ?), that executes fixed-length, 32-bit instructions.
- It is actually a simplified version of **MIPS**, the first RISC architecture (to us, straightly a *modern architecture*), designed by J. Hennessy in the early 80's.
- To begin with, we shall consider the "integer" section of the architecture only: instructions operate on integer registers (add, sub, etc.), memory (load, store) and PC (branches, unconditional and conditional).

A simple RISC architecture

- We shall cover mainly the **datapath** of the CPU, with only some hints to the **control unit**: the section that controls and synchronizes the different datapath blocks during instruction execution.
- We shall however insist on two main features of a modern RISC machine:

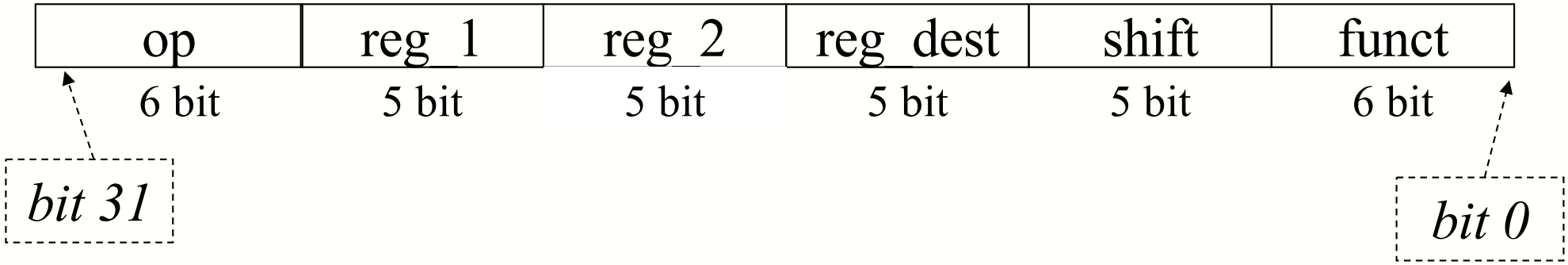
1. machine instructions are *simple*;
2. as a consequence, the control unit is simple too, and there is no need for a complex microprogram to steer machine instruction execution within the CPU.

A simple RISC architecture

- The architecture has (at least) 32 general purpose integer registers, each 32-bit in depth. In MIPS, as well as in most machines, register R0 is special and always stores 0 (the same is true of 64-bit architectures).
- Since we do not consider floating point operations, the architecture we describe has neither floating point registers nor **functional units** (datapath blocks) to process floating point data.

MIPS instruction format

- A generic MIPS instruction has the following **format**:



- op: class of operation
- reg_1: source register 1
- reg_2: source register 2
- reg_dest: destination register
- shift: used in shift ops on registers
- funct: specifies the type of the operation within its class

Fields in MIPS instructions

- With these fields we can describe the following format:
- **R-type instructions**: have one or two register operands and produce the result in a third register. As an instance:
  - DADD R1, R20, R30 // R1 ← [R20] + [R30]

0	20	30	1	not-used	32

  - DSUB R5, R11, R12 // R5 ← [R11] – [R12]

0	11	12	5	not-used	34

Fields in MIPS instructions

- A few notes:
- a "D" prefix usually denotes an integer instruction (one which acts on integers); "F" is used for floating point.
- In the symbolic notation, the first register is the destination register.
- In the two instructions just considered, "op = 0" denotes only that both operations use the **ALU**, act on two registers and store the result in a third one. It is the *funct* field that specifies the exact operation (an add, a sub, a multiply, and so on).

Fields in MIPS instructions

- Instruction fields can have other functions. In **I-type instructions**, that use an *immediate* as operand (an integer), the layout of fields is the following:

op	reg_1	reg_dest	immediate value

- As an instance:
  - DADDI R5, R1, 1500 // R5 ⟵ [R1] + 1500

8	1	5	1500

- The *op* fields takes on a different value, and the immediate value is represented with 16 bits.

Fields in MIPS instructions

- This format can be used also for **load/store. instructions.** As an instance:
  - LD R1, 128(R4)

35	4	1	128

  - loads into R1 4 bytes from main memory, starting from the byte at address 128 + [R4]
  - SD R1, 252(R4)

43	4	1	252

  - stores the content of R1 in memory, starting at the address 252 + [R4]

Fields in MIPS instructions

The final format includes:

- **branch instructions (conditional):**

op	reg_1	reg_2	offset

as an instance: BNE R1, R2, 100 // jump if R1<>R2

5	1	2	25

- **jump instructions (unconditional) :**

op	offset

as an instance: JMP 10000

4	2500

Fields in MIPS instructions

- Note that we are skipping a lot of details. As an instance, what about conditional jump addresses larger than 2^{16} bytes (this is indeed important, can you tell why?)
- What about procedure calls ?
  - (*usually, a subset of the registers – 4 in MIPS – are used for parameter passing, another subset – 2 in MIPS – for the result*)
- What about 32-bit constants ?
- These items are basic issues, all to be addressed and solved in any real implementation, but not mandatory to understand the operation of a modern CPU.

A single cycle version of MIPS

- MIPS instructions execution is similar for any type of instruction (this is true of all RISC architectures).
- The first two steps are actually always identical in any instruction:

  1. the PC is used to fetch from memory instruction the instruction to be executed (Instruction Fetch)
  2. Instruction decoding and reading one or two registers, using the proper instruction fields to select the register(s)

A single cycle version of MIPS

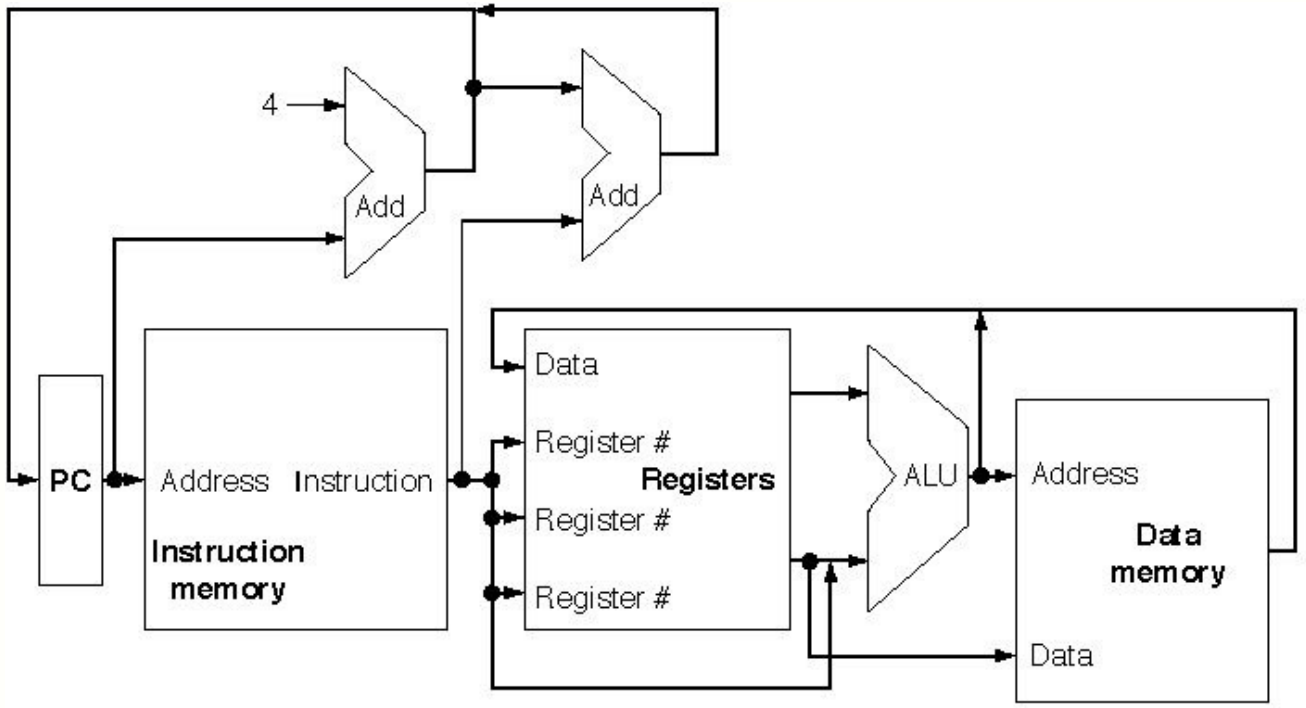
- After the first two steps, the subsequent actions depend on instruction type:
  - memory access
  - arithmetical-logical operation
  - branch/jump
- nethertheless, actions are almost the same within each instruction type, and different instructions types share many actions.

A single cycle version of MIPS

- As an instance, all instructions (except jump) use the ALU.
- Instructions are completed in different modes :
  - Load and Store access data memory, and Loads updates a register
  - Arithmetical-logical instructions update a register
  - Jump/branch instructions change (conditionally, in branches) PC value

A single cycle version of MIPS

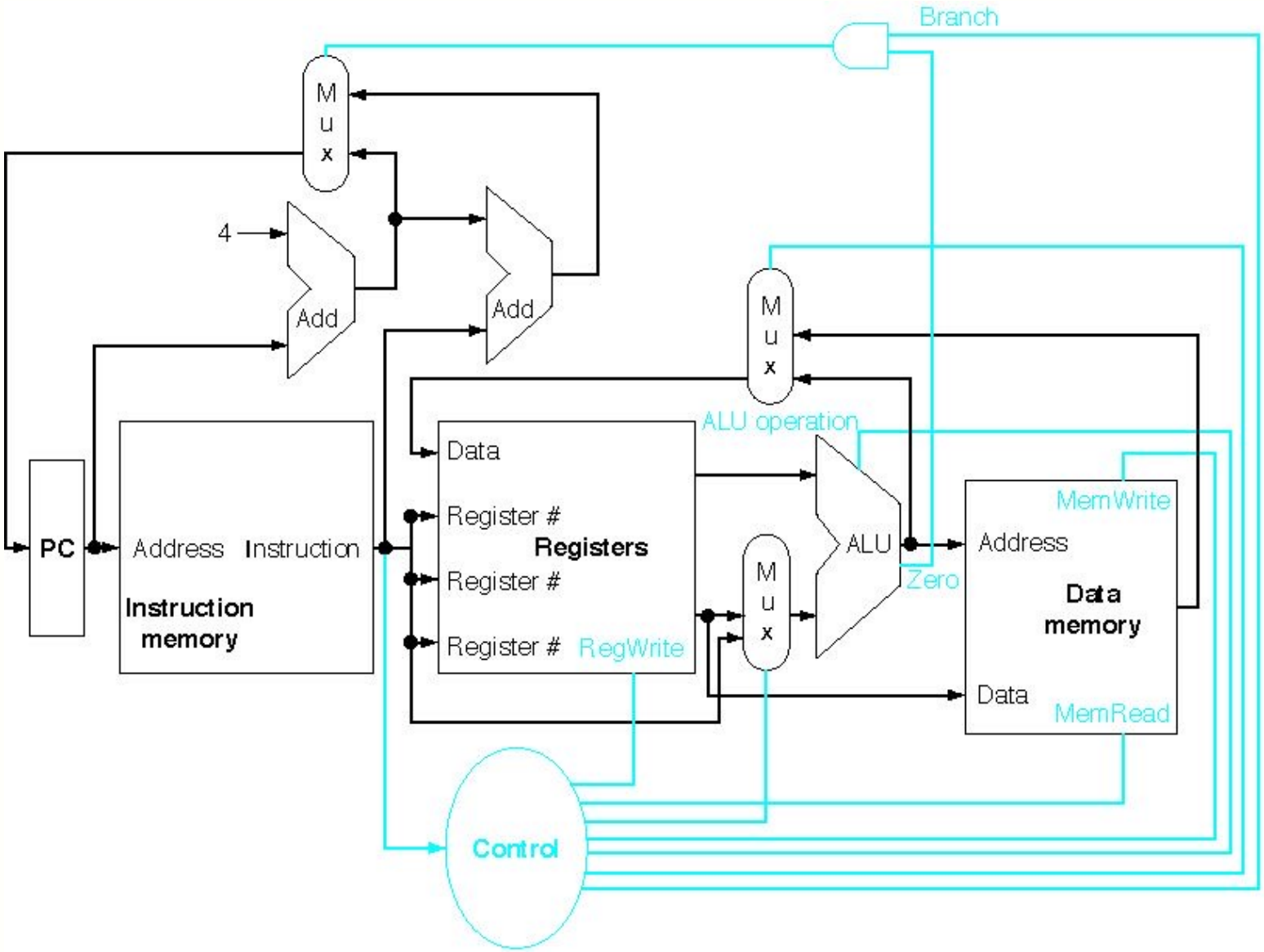
- Here is a high level sketch of MIPS datapath, highlighting the main functional units and their interconnections. Note the separate memory units for instructions and for data. More on this in the following.
(Patterson-Hennessy, fig. 5.1)



A single cycle version of MIPS

- In the following drawing: a high level schemeof MIPS datapath, with the control unit, the multiplexers to select the correct inputs, and the signals to control functional units and muxes.
  (Patterson-Hennessy, fig. 5.2)
- Get acquainted with this high level scheme (and the following ones), that is common to all modern architectures (in its basics blocks, of course).

A single cycle version of MIPS



A single cycle version of MIPS

- Each instruction can be executed in this datapath in a single clock cycle: the clock cycle has to be long enough for each instruction to flow through all sections of the datapath it requires for its execution.
- To understand its operation, let us recall that the datapath of any processors is made up of two types of logical elements (we refer to them as a whole as functional units): **state** and **combinational** elements.

A single cycle version of MIPS

1. **state elements** allow to store a value, such as a flip-flop, and, in the MIPS datapath, registers and memories.

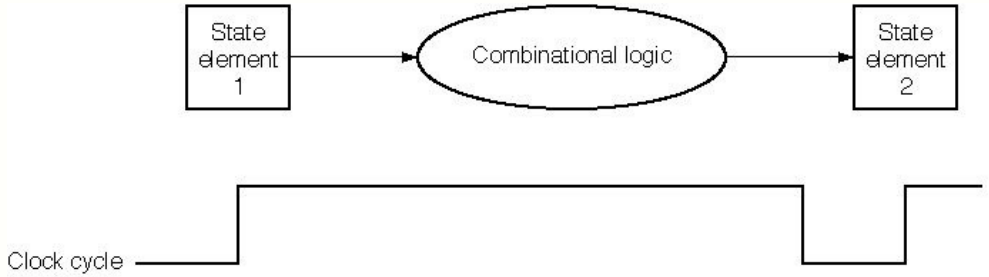
- a state element has at least two inputs and one output. Inputs include : 1) the value to be stored and 2) the clock, that establishes when the value is actually stored (usually, on the rising edge / falling edge of the clock cycle).

- at any time, the value available at the output of a state element is the one stored in the preceding clock cycle.

A single cycle version of MIPS

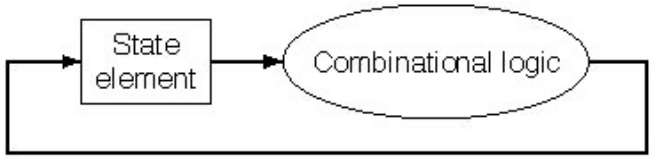
2. **combinational elements**: the outputs depend only on the values available at the inputs at a given instant (one must allow for some propagation delay through the logical gates that make up the element), such as the ALU the muxes.

- At a higher abstraction level, the single cycle execution of an instruction within the datapth can be depicted as in the following picture: (Patterson-Hennessy, fig. B.7.2)



A single cycle version of MIPS

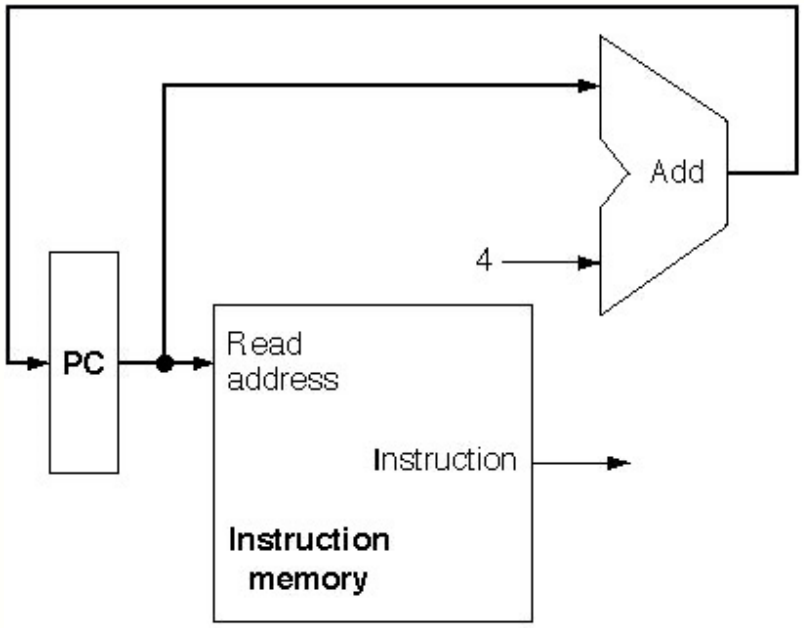
- For a correct operation, it is necessary that the clock cycle be long enough so that inputs to a state element become stable before the clock edge stores them into the element
- Note that two state elements can coincide: (Patterson-Hennessy, fig. 5.4).



- In this case, during the first part of the clock cycle, the state element transfers the value to the combinational logic (*reading* phase). This in turn produces an output that will be stored in the same element during the second part of the same clock cycle (*writing* phase).

A single cycle version of MIPS

- In MIPS datapath, an instance is the increment of the Program Counter at each new clock cycle, with the help of a dedicated ALU. The PC is used in every cycle to fetch from the instruction memory the next one to be executed. (Patterson-Hennessy, fig. 5.6)

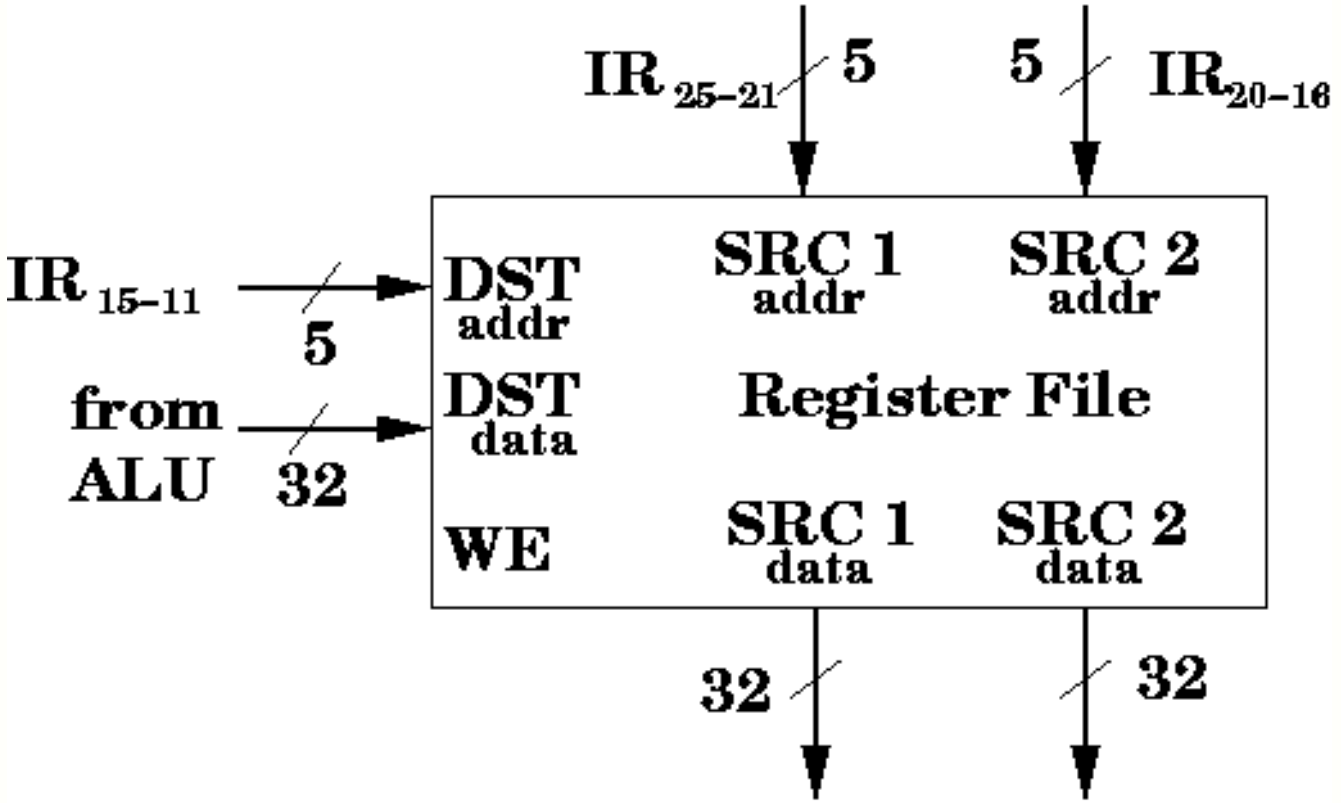


The Register File

- In the datapath we are considering, the CPU *general purpose* registers are depicted as a single functional unit called “Registers”.
- Actually, these registers are assembled in a structure made up of actual memory units (the registers) and of a control logic, that allows to access each register by specifying its number and the type of access (read or write).
- This structure is called **register file**. It works as a normal memory bank, internal to the CPU, very small and extremely quick.

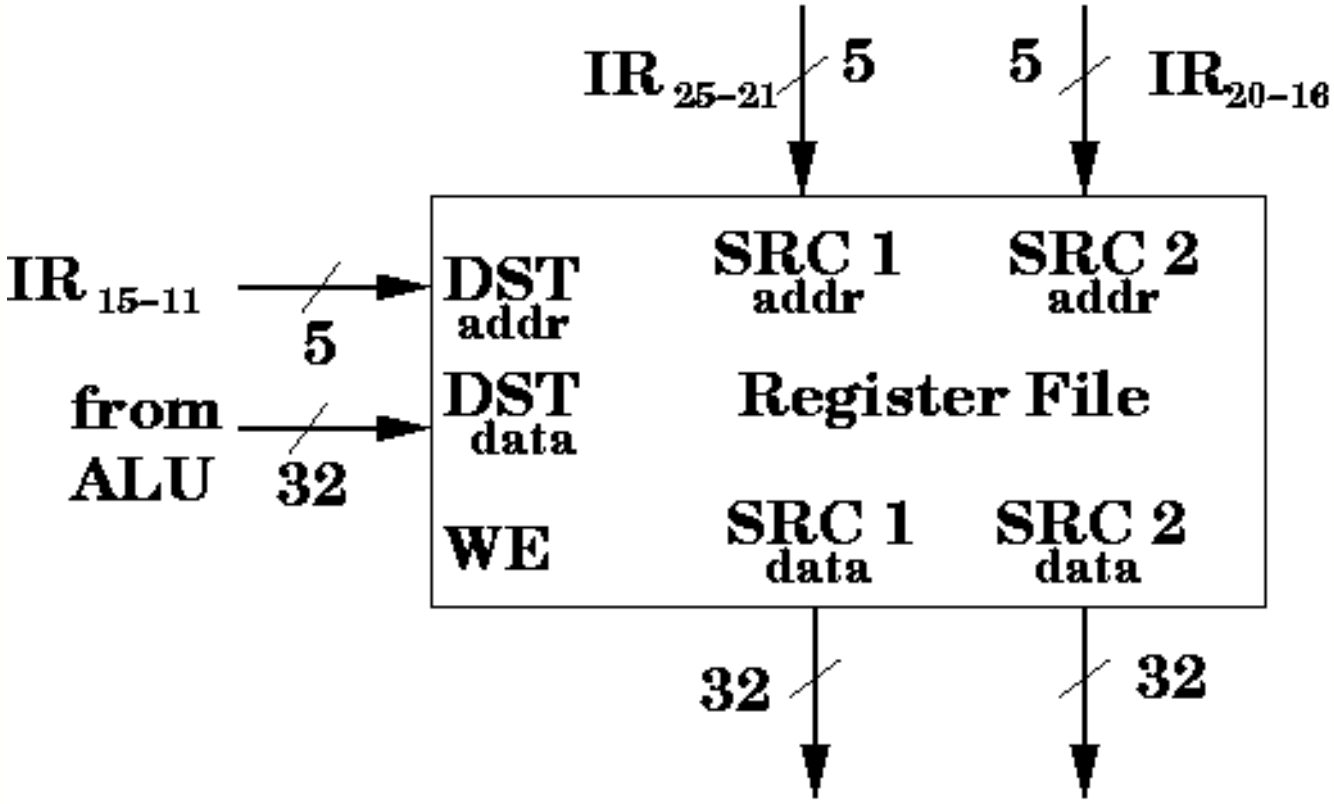
The Register File

- The external interfaces of the register file. The WE (*Write Enable*, *RegWrite* in our datapath) control signal is output from the control unit. If asserted, it allows to write the output from the ALU (DSTdata) into the destination register (DSTaddr) specified in the instruction.



The Register File

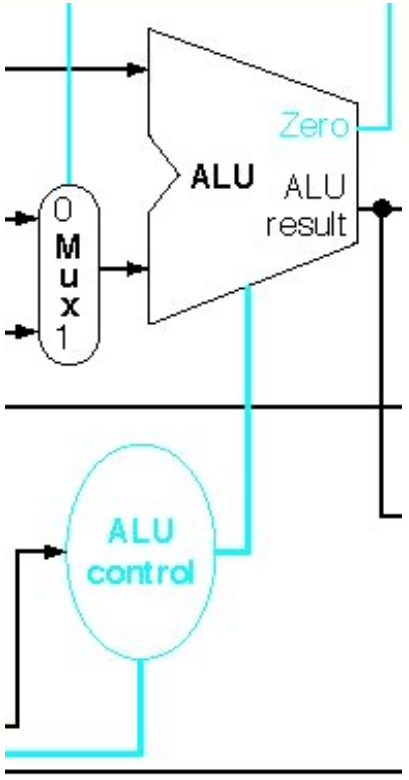
- Write operations into the register file are controlled by a specific signal; read operations are “immediate”: at any time, the register file outputs on *SRC1data* and *SRC2data* the content of registers addressed through inputs *SRC1addr* and *SRC2addr*



A simple Control Unit

- To set up a CU (Control Unit) in MIPS it is necessary to define the control signals for the ALU, that receives two inputs (which ones?) and transforms them into an output according to four control signals (an hypothetical scheme could be):

ALU control	Operation
0000	AND
0001	OR
0010	addition
0110	subtraction
0111	set on less than
...	...



A simple Control Unit

- Control signals for the ALU can be generated by a simple ALU control sub-unit (a part of the CU) that receives in input:
  - the “funct” field of the executing instruction
  - two control bits “ALUop” from the CU, that depend on the op field of the instruction, and establish if the operation to be carried out is a sum for a load or store, a subtraction, or is further chosen on the basis of the “funct” field.

op	reg_1	reg_2	reg_dest	shift	funct

A simple Control Unit

- Patterson-Hennessy, fig. 5.12:

ALUcontrol OUTPUT

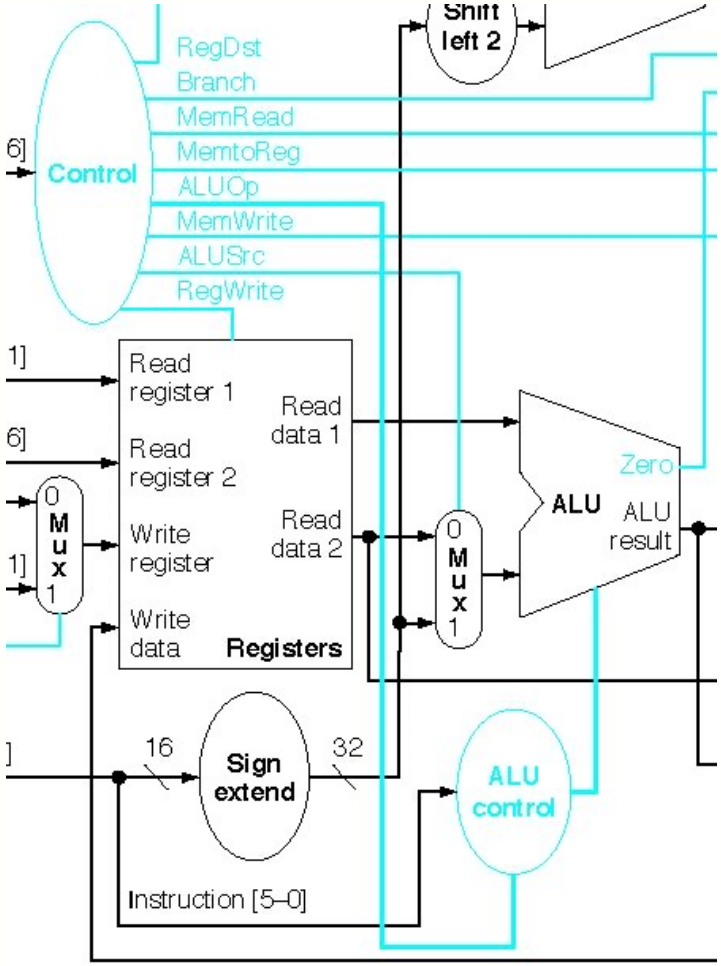
ALUcontrol INPUT

Op	ALUop	operation	funct	ALU function	ALU control
load	00	load word	xxxxxx	sum	0010
store	00	store word	xxxxxx	sum	0010
branch eq.	01	branch eq.	xxxxxx	subtraction	0110
tipo-R	10	somma	100000	sum	0010
tipo-R	10	sottrazione	100010	subtraction	0110
tipo-R	10	AND	100100	and	0000
tipo-R	10	OR	100101	or	0001
tipo-R	10	set on less than	101010	set on less than	0111
...	...		...	...	...

op	reg_1	reg_2	reg_dest	shift	funct

A simple Control Unit

- ALUop bits are produced by the CU and are used as input (with "funct" bits) to the ALU control sub-unit



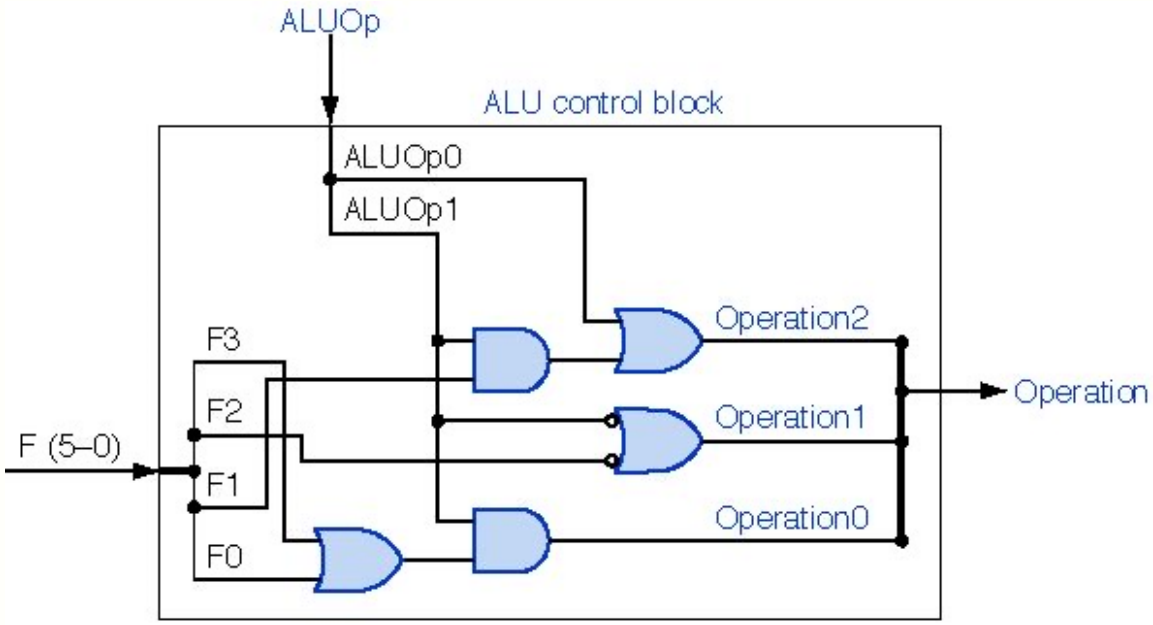
A simple Control Unit

- ALUcontrol is the logic circuit that realizes the truth table that maps inputs bits ALUcontrol (ALUop + funct) into the outputs, the 4 ALU control bits (Patterson-Hennessy, fig. 5.13):

ALUop0	ALUop1	F5	F4	F3	F2	F1	F0	ALUcontrol
0	0	x	x	x	x	x	x	0010
x	1	x	x	x	x	x	x	0110
1	x	x	x	0	0	0	0	0010
1	x	x	x	0	0	1	0	0110
1	x	x	x	0	1	0	0	0000
1	x	x	x	0	1	0	1	0001
1	x	x	x	1	0	1	0	0111
...	...	...	...	...	...	...	...	...

A simple Control Unit

- Here is ALUcontrol (empty dots negate inputs to the OR gate). The two most significant bits in "funct" are not used (x stands for "don't care"). Patterson-Hennessy, fig. C.2.3.

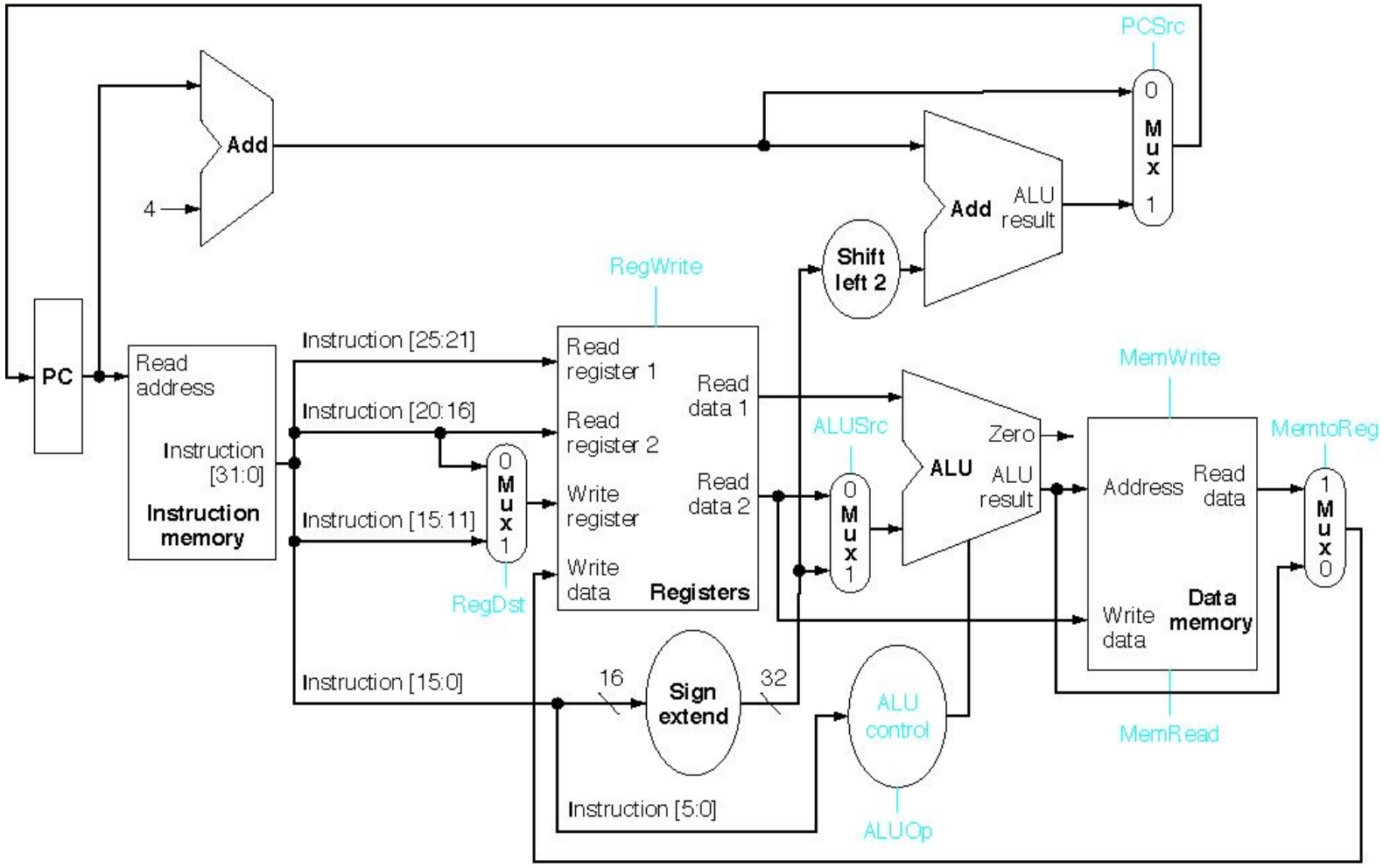


A simple Control Unit

- The design of the main CU follows the steps outlined for ALU control.
- The main Control Unit has in input the 6 bits of the “op” instruction field (bits 31-26) and has to output control signals for :
  - enabling registers write (*RegWrite*)
  - controlling data memory read and write (*MemRead* and *MemWrite*)
  - setting the muxes (*RegDst*, *ALUSrc*, *MemtoReg*, *PCsrc*)
  - controlling the ALU (*ALUop*, the two signas just discussed)

A simple Control Unit

- The MIPS datapath with all signals generated from the CU. Note the new mux for write selection at the inputs of the register file (Patterson-Hennessy, fig. 5.15)



A simple Control Unit

- In the next chart, a table listing the purpose of each control signal (excluding *ALUOp*, already described), with the action resulting when asserted (set to 1) and when left unasserted (set to 0). Patterson-Hennessy, fig. 5.16.

signal	action if unasserted	action if asserted
RegDst	the destination register number for *Write register* comes from field reg_2 (bits 20-16)	the destination register number for *Write register* comes from field reg_dest (bit 15-11)
RegWrite	none	the value at input *Write data* is stored into the register selected with *Write register*
ALUSrc	the second ALU operand comes from the second output from the register file	The second ALU operand comes from the 16 low order instruction bits
PCSrc	PC is loaded with the ouput from ADDER that computes PC+4	PC is loaded with the output from the ADDER that computes the jump destination
MemRead	none	The content of memory location addressed with *Address* is put on *Read data* output
MemWrite	none	The value on *Write data* is stored in memory location addressed with *Address*
MemtoReg	The value on *Write data* input at register file comes from ALU	The value on *Write data* input at register file comes from data memory

A simple Control Unit

- The Control Unit is simply a combinational circuit, actually a truth table, with inputs from the 6 bits of the "op" field of each instruction, and 9 outputs, namely the control signals in the MIPS datapath.
- A section of this truth table (for R-type MIPS instructions, load, store and beq (branch if equal). Patterson-Hennessy, fig. 5.22.
- In actual implementations, there are more input/output combinations, for all other instructions in MIPS ISA.

A simple Control Unit

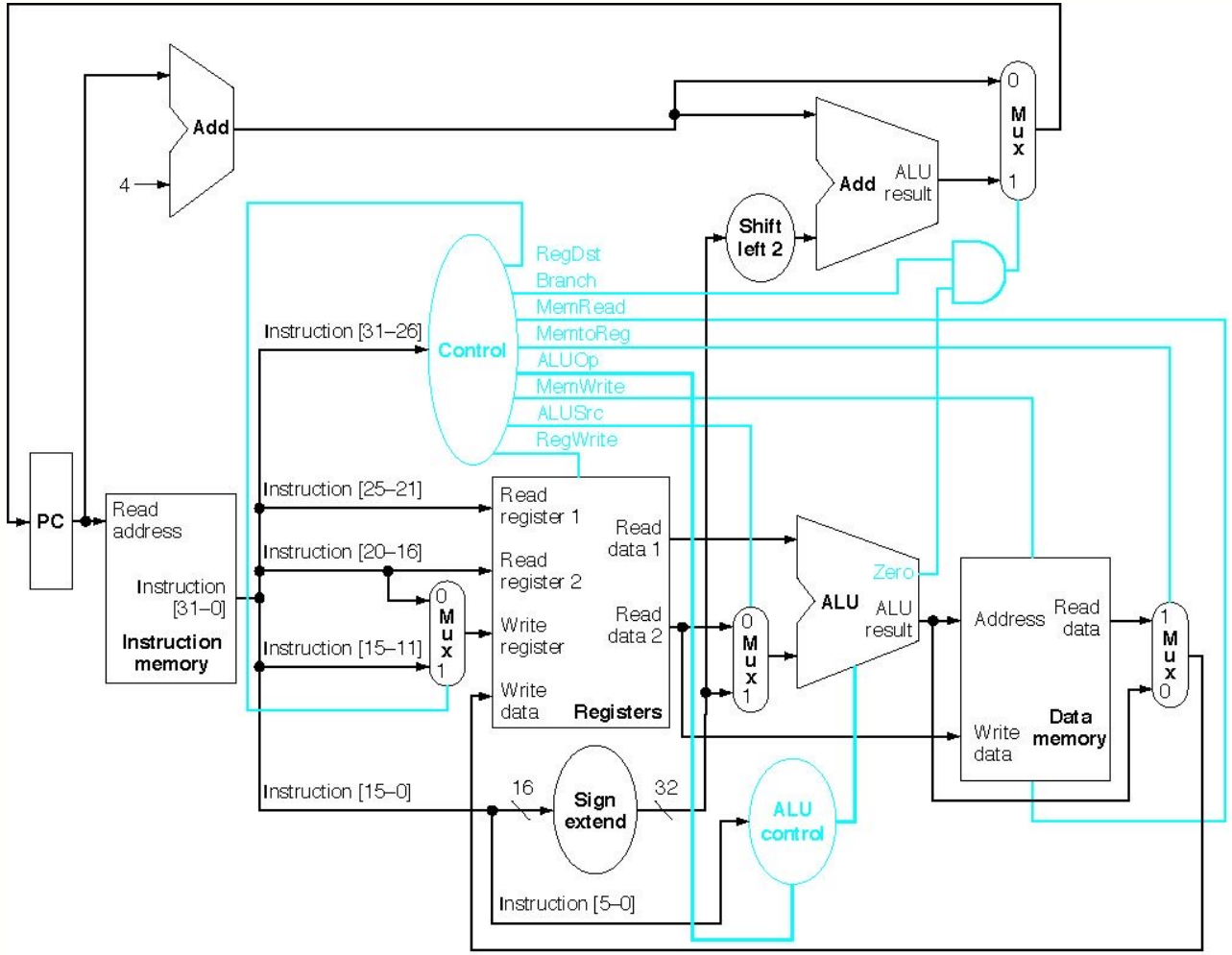
input

otput

signal	R-type	load	store	beq
op5	0	1	1	0
op4	0	0	0	0
op3	0	0	1	0
op2	0	0	0	1
op1	0	1	1	0
op0	0	1	1	0
RegDst	1	0	X	X
ALUsrc	0	1	1	0
MemtoReg	0	1	X	X
RegWrite	1	1	0	0
MemRead	0	1	0	0
MemWrite	0	0	1	0
Branch	0	0	0	1
ALUOp1	1	0	0	0
ALUOp0	0	0	0	1

A simple Control Unit

- The MIPS datapath and its Control unit. Note the AND gate to choose the source for the next PC value (Patterson-Hennessy, fig. 5.17)



A simple Control Unit

- As an instance, let us track the execution of an R-type instruction.

1. The PC content is used to address the Instruction Memory, which outputs the instruction to be executed.

2. The instruction *op* field is sent to the Control Unit, and fields *reg_1* and *reg_2* are used to address the register file.

- Please note that at this stage, the datapath has no notion of the instruction type (R-Type, or whatever), namely that the required values come from *reg_1* and *reg_2* registers; more on this subject later on …

A simple Control Unit

3. The CU produces nine control signals for an R-type instruction, specifically *ALUOp*=10; these, once input to ALUcontrol with the *funct* instruction field, set the actual R-type operation to be carried out by the ALU

4. Meanwhile, the register file outputs the values of registers *reg_1* and *reg_2*, as signal *ALUSrc*=0 selects the second input to the ALU from the second output of the register file.

5. The ALU outputs the result from the computation, which is forwarded to the register file (Write data) according to signal *MemtoReg*

A simple Control Unit

- All these steps can be carried out within the same clock cycle, provided the clock cycle is longer than the cumulative signal delay through all functional units involved.
- So far, datapath state elements involved in the execution (PC, instruction memory, register file) have not changed their state (they have stored no new value).
- They just behaved as combinational elements, delivering as ouputs the values at their inputs, with no delay.

A simple Control Unit

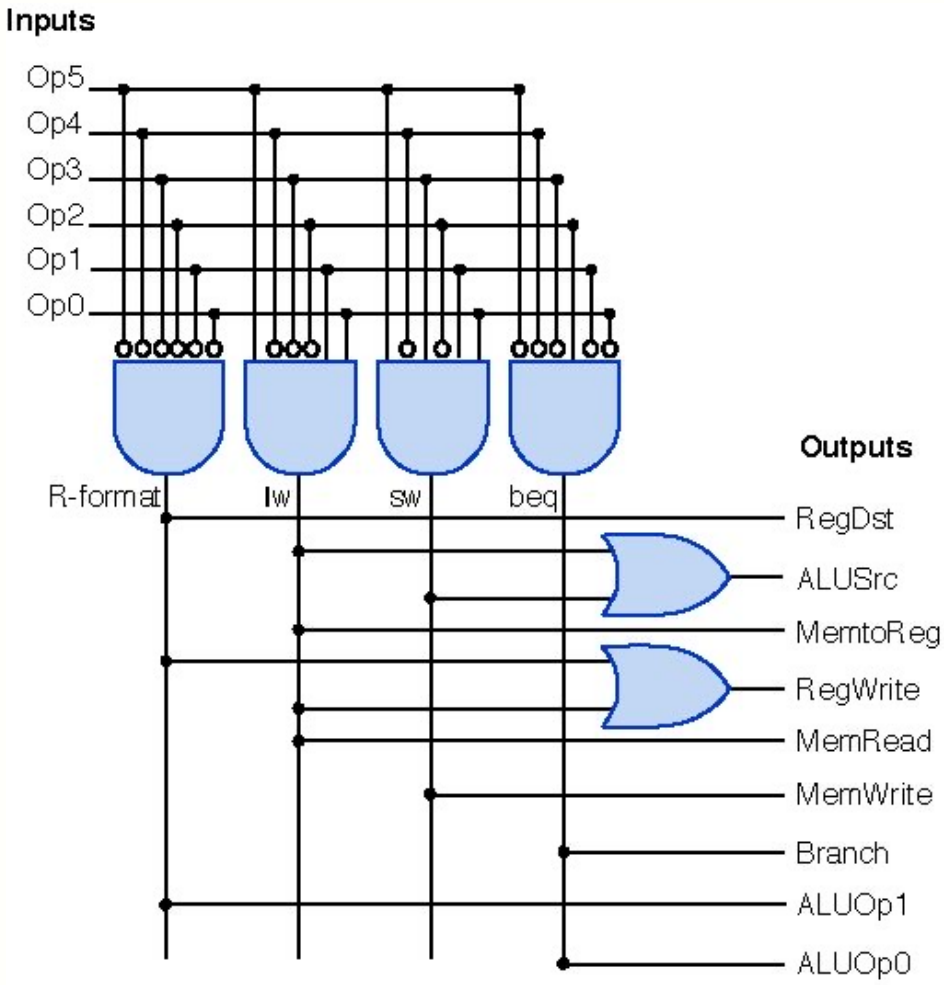
6. At this point, the value available at register file input *Write data* has to be stored into the register addressed with the instruction field *reg_dest*, and the actual store happens at the proper clock transition, according the flip-flop type.

- Note that control sugnal *RegWrite* is asserted, to enable the write into the destination register of the register file.

- As an exercise, repeat the sequence just analysed for the other instruction types (load, store, beq), checking each control signal.

A simple Control Unit

- Here is the Control Unit for the single cycle architecture.
(empty dots negate signals at AND gate inputs)
Patterson-Hennessy, fig. C.2.5.



A multicycle version of MIPS

- Single cycle datapaths can work correctly, but they are not used in current implementations, because of their pour efficiency.
- Different instructions take a different amount of time to be executed, according to the required operations within the datapath.
- The clock cycle must be chosen once for all, and its duration must accommodate for the longest instruction; this in turn causes a waste in all instructions that executue in a shorter time.

A multicycle version of MIPS

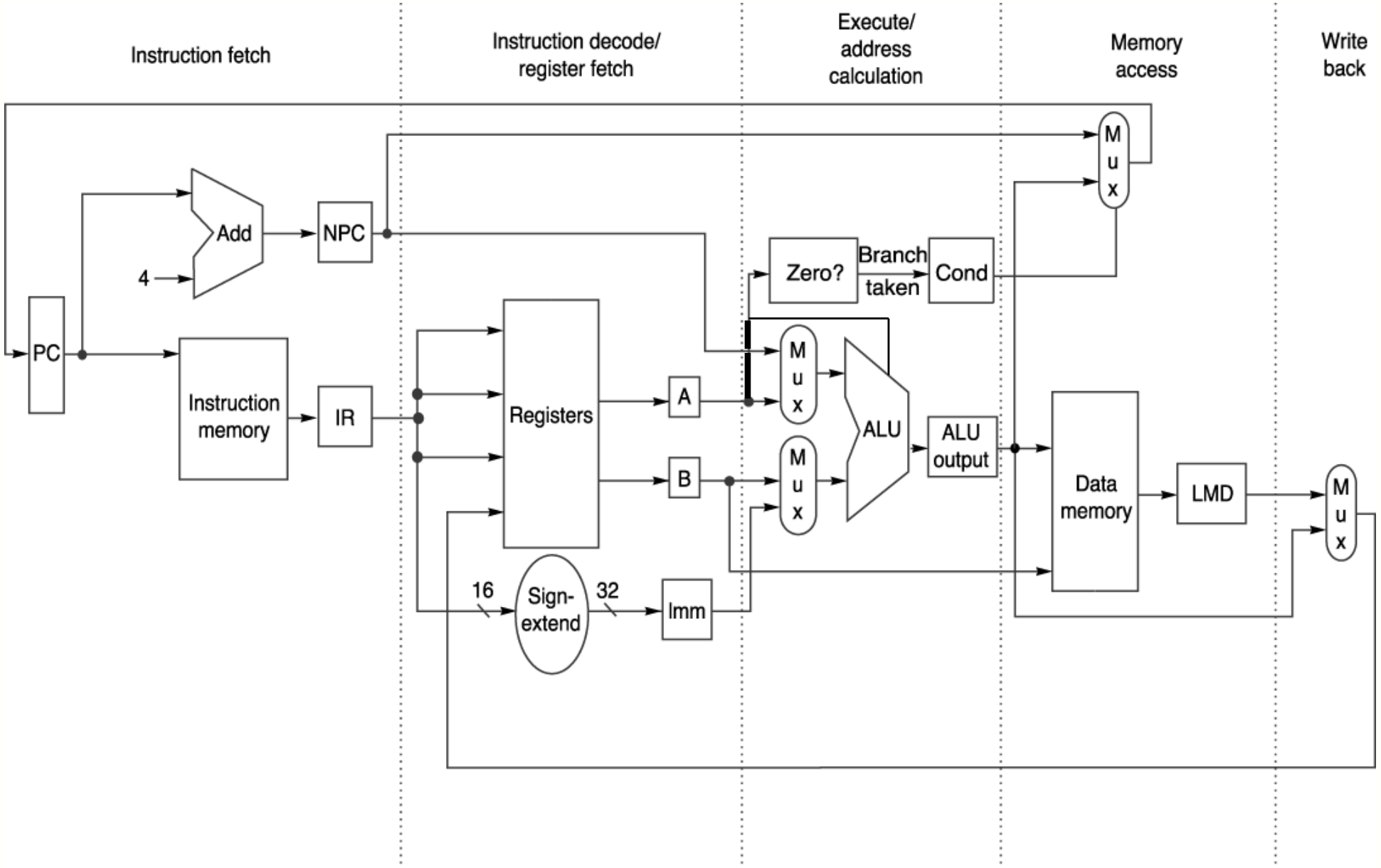
- Which are the design criteria for a modern multicycle processor?
- As a first step, the designer breaks instruction execution down into a set of steps, each of them deemed executable in a single clock cycle (Note: in the following, we will use interchangeably the terms *step*, *phase*, *stage*)
- It is highly advisable that each step can be carried out in roughly the same amount of time. Indeed, if a step is longer than the others, the clock cycle must match the longest !

A simple RISC architecture

- Each stage can host at most a single operation on each functional unit.
- If the CPU hosts a single ALU, it cannot be used in the same step both to increment the PC and to add two registers
- Let us inspect the complete datapath of the multicycle MIPS, just to comment its operation at each step of the execution of the instructions already considered.

The MIPS architecture datapath

Hennessy-Patterson, Fig. A.17:



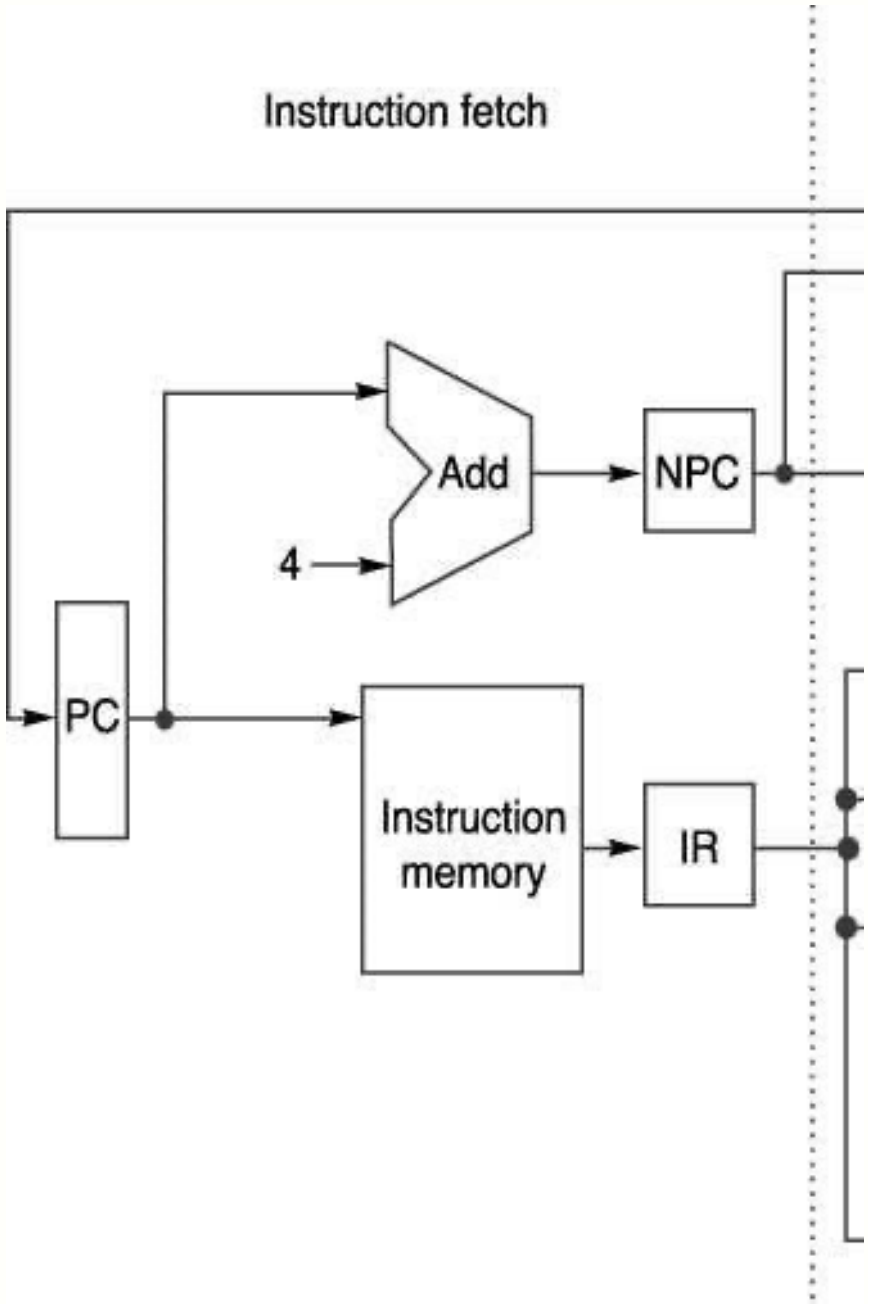
MIPS: Instruction Fetch

1. First step: Instruction Fetch (IF):

a) IR ‹- memory[PC]:
uses PC to fetch from instruction memory the instruction to be executed, and stores it in the Instruction Register (IR).

b) PC ‹- PC + 4.

- As we know, a) and b) are executed in parallel.

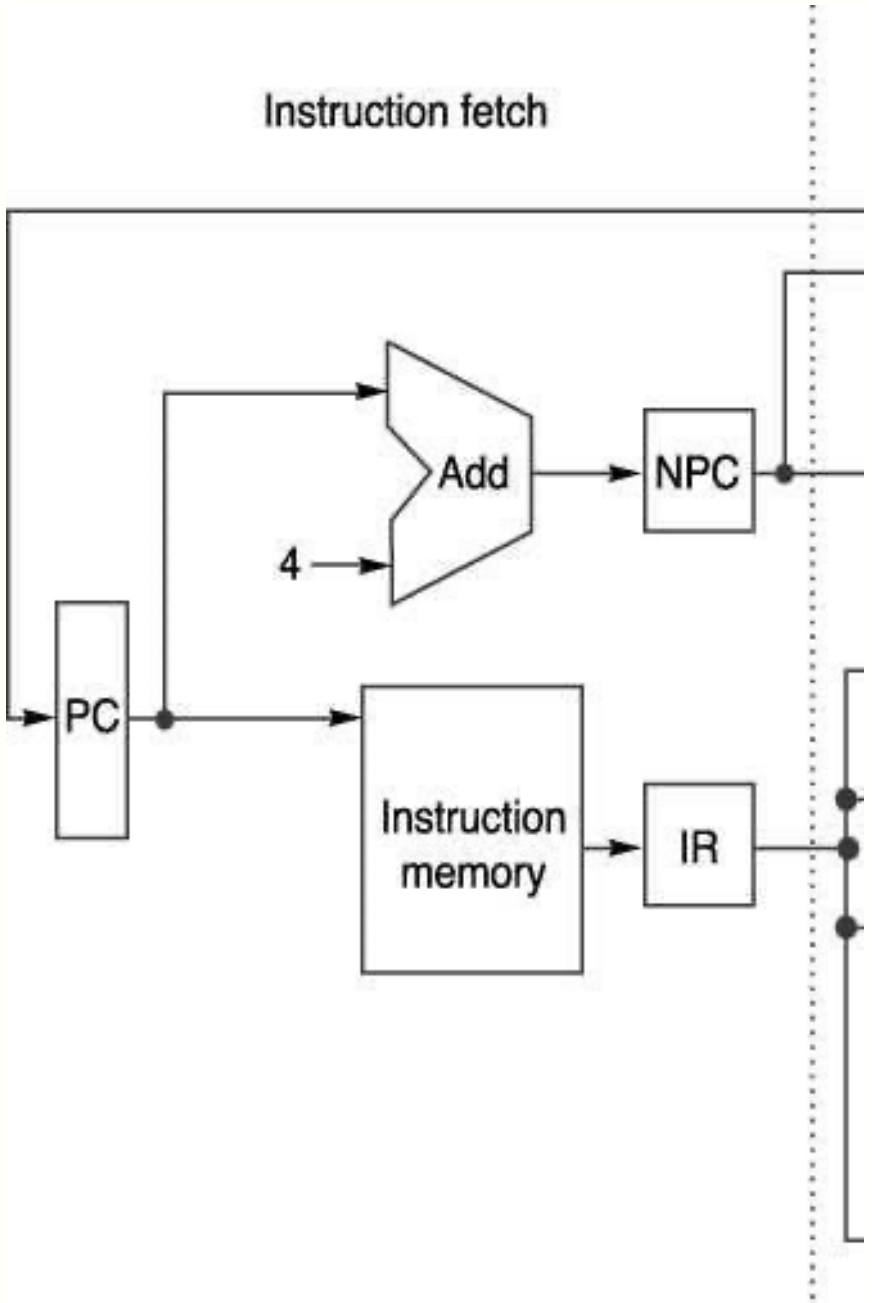


MIPS: Instruction Fetch

- Let us immediately highlight some features of the architecture operation.
- The PC increment could be done by the ALU, that is not used in this phase of the instruction execution.
- But when we will consider instructions pipelining, we will note that the ALU is (hopefully) busy, with another instruction.
- It is better to have a separate ADDER for the PC alone.
- Also, please note that the new PC (NPC) is also forwarded towards the ALU (can you tell why ?)

MIPS: Instruction Fetch

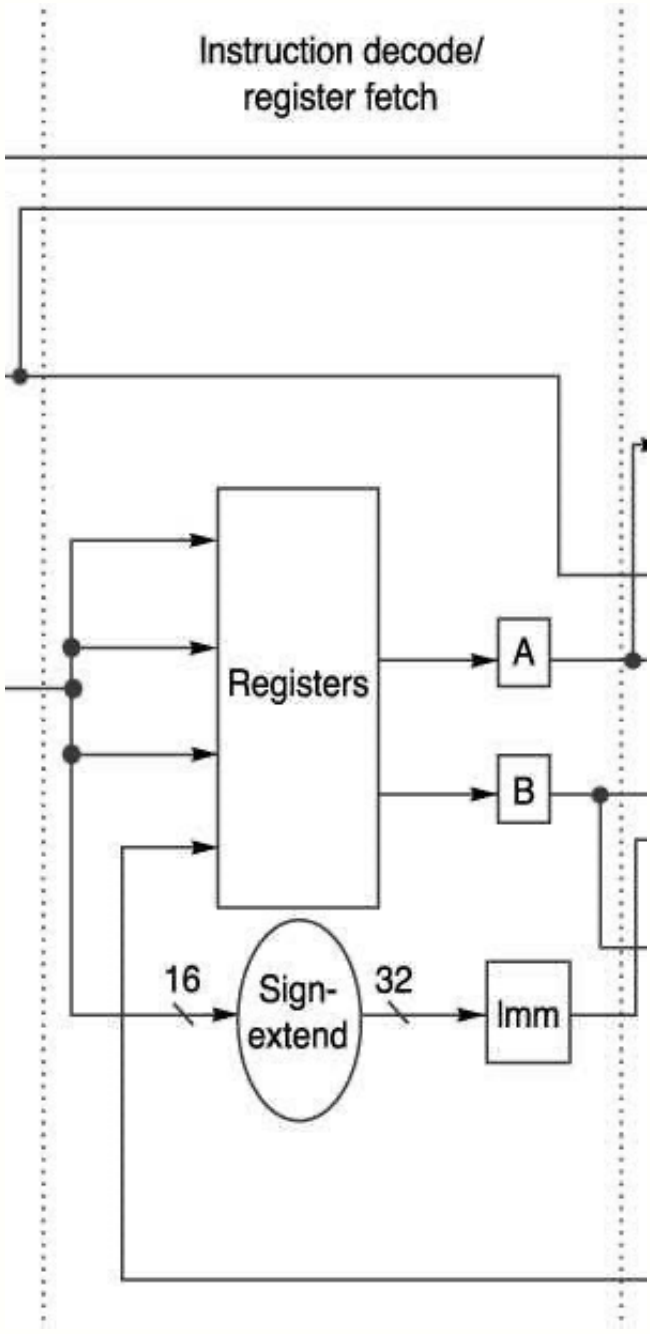
- At the end of this phase, the new PC and the instruction are stored in two internal registers (NPC e IR).
- These registers are not visible at the ISA level (so, the programmer cannot use them), yet they are necessary in the multicycle implementation to store mandatory data required in the following phases



MIPS: Instruction Decode

2. Instruction decode/register fetch (ID)

- At the begin of this phase, the instruction type is still unknown; however, it is possible to carry out operations that cause no damage, if the decoding shows a different instruction.



MIPS: Instruction Decode

a) A ‹- reg[IR[25-21]]
B ‹- reg [IR[20-16]]

- reading of source registers. It is still unknown if the registers will be actually used (R-type instruction) but reading them causes no harm, and they could turn out to be useful in later steps, so they are read from the register file and stored in temporary registers A and B (invisible at ISA level).
- The register file is read at every cycle, and temporary registers A and B are overwritten at every cycle too.

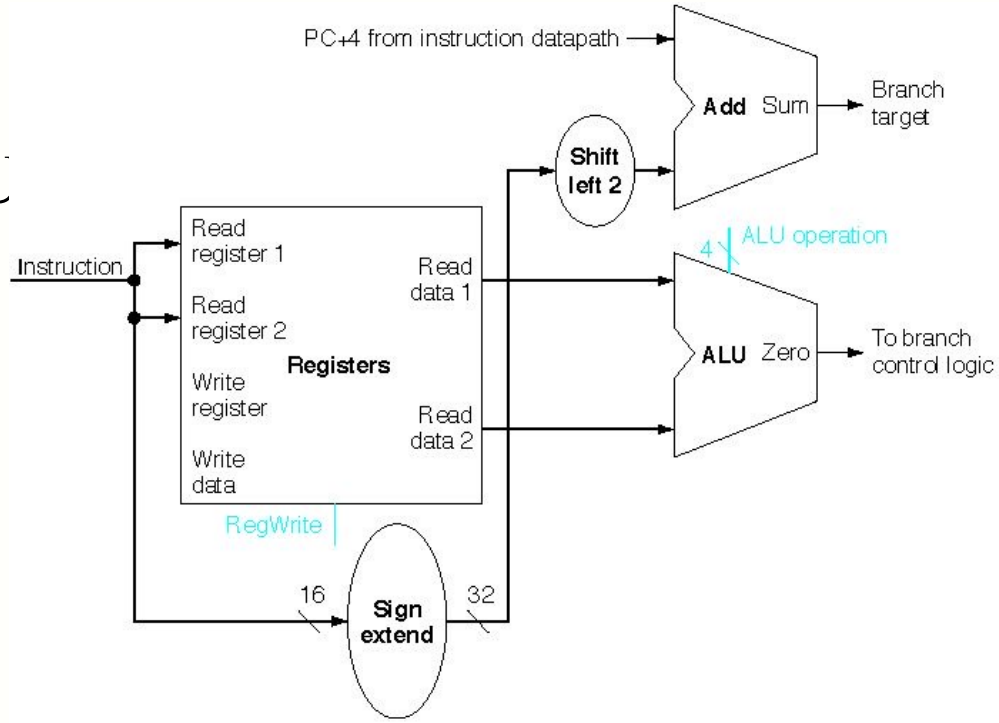
MIPS: Instruction Decode

b) ALUout ‹- PC + IR[15-0]

- the hypothetical branch destination address is computed and stored in the internal register ALUoutput, whence it will be fetched in the next cycle if the instruction is indeed a branch (Patterson-Hennessy, fig. 5.9).

Note (1): this picture shows an ADDER for computing PC+IR[15]. This could be done through the ALU (go back to the complete datapath scheme)

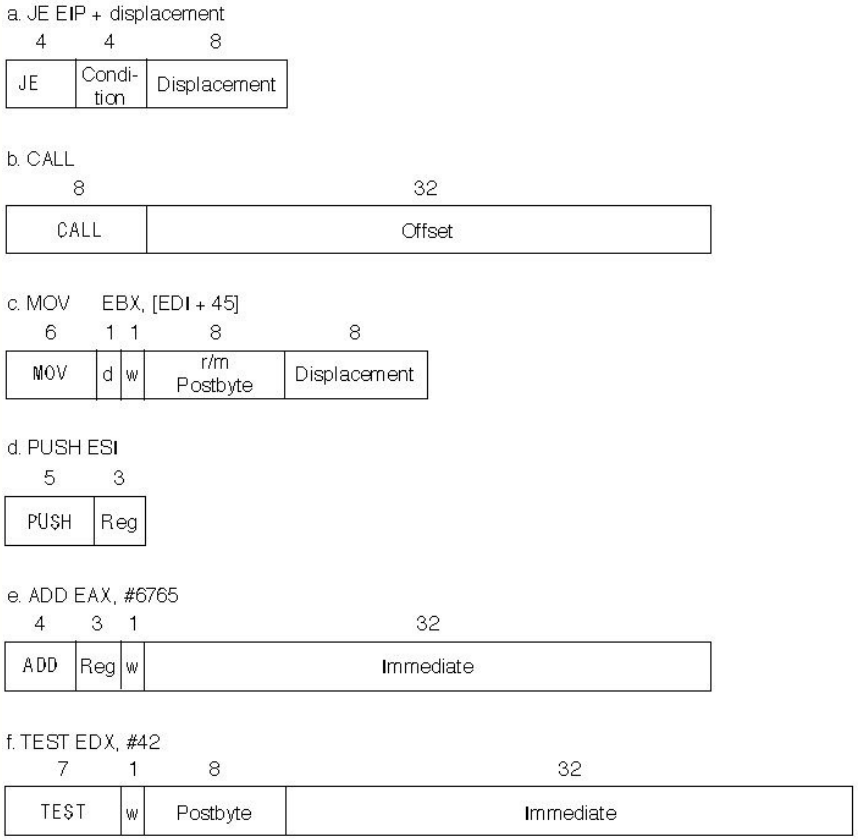
Note (2): The true increment is :
ALUout <u>‹-</u> PC +
+ (sign-extended (IR[15-0]) << 2



MIPS: Instruction Decode

- In an “aggressive” implementation, the test on the branch could be carried out during the second clock cycle, thus completing a branch instruction in two cycles.

- If the instructions have an irregular structure, (as in IA-32), it is not possible to identify the bits for the operands until the instruction is decoded (Patterson-Hennessy, fig. 2.45)



MIPS: EXecution

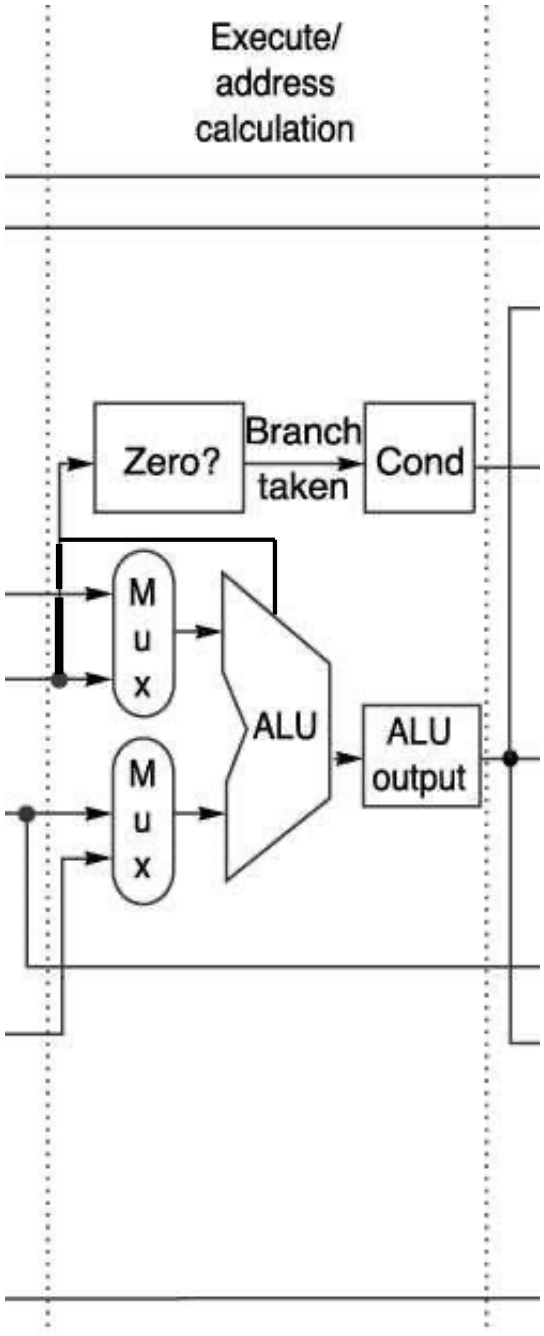
3. **Execution (EX):** The Control Unit drives the ALU using the op-code in the instruction register, and the ALU works on the operands (available from the previous cycle) by executing one of four possible operations, according to the instruction type:

   a) **I-type instruction, memory access (load and store)**

   b) **R-type instruction (register-register)**

   c) **conditional branch**

   d) **unconditional branch (jump)**



MIPS: EXecution

a) I-type instruction or memory access (load/store)

- ALUout ‹- A + IR[15-0]
- the ALU adds to tempoary register A the immediate value (in a load/store, this is the RAM address to work on)

b) R-type instruction (register-register)

- ALUout ‹- A op B
- the ALU executes the specified operation on the contents of temporary registers A and B

MIPS: EXecution

c) Conditional branch

- if (A == B) PC ‹- ALUoutput
- The ALU compares the two source registers, and signal Zero, if asserted, has the content of ALUoutput stored in the PC.

d) Unconditional branch (jump)

- PC ‹- jump address
- Note that in a *taken branch*, the PC is written twice: once during the IF phase, then it is overwritten during EX

MIPS: MEMory access

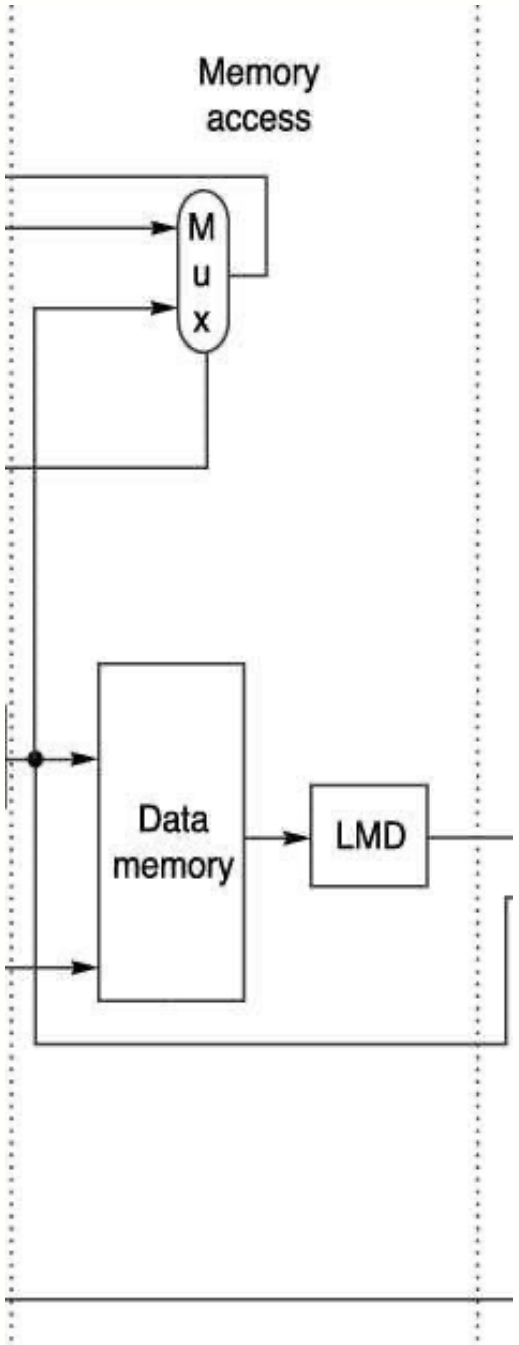
4. Memory access (MEM/WriteReg):

a) load/store

- MDR ‹- memory[ALUoutput] (load)
- memory[ALUoutput] ‹- B (store)

b) arithmetic/logical instruction (I or R-type)

- Reg[IR[15-11]] ‹- ALUoutput



MIPS: MEMory access

- In MIPS architecture datapath, it looks like there are two memories, one for the instructions, accessed during "Instruction Fetch", another for the data, accessed during "MEMory access".
- They could also be realized as a single memory, since the two memories are accessed in different phases of instruction execution (that is, in different clock cycles).
- This is the well known principle of operation of modern computers: the RAM stores both data and instructions.

MIPS: MEMory access

- However, when considering instruction pipelining, it will be clear that both memories are accessed in the same clock cycle, because of the two instructions that are both executing, though in different phases: were they not distinct, the concurrent access would be impossible.
- Indeed, in MIPS datapath, as well as *in any other modern CPU*, the Instruction Memory and the Data Memory **are two distinct memories**.

MIPS: MEMory access

- When we discuss caches, we'll see that these memories are actually *first-level caches*; the first level cache is split into I-cache (instruction cache) and D-cache (data cache), with separate addrressing and data lines.
- This double memory, located within the CPU, works at the same speed as the CPU; so it does not delay instructions execution.
- Things get worse when the addressed data or instruction are not found within the cache...

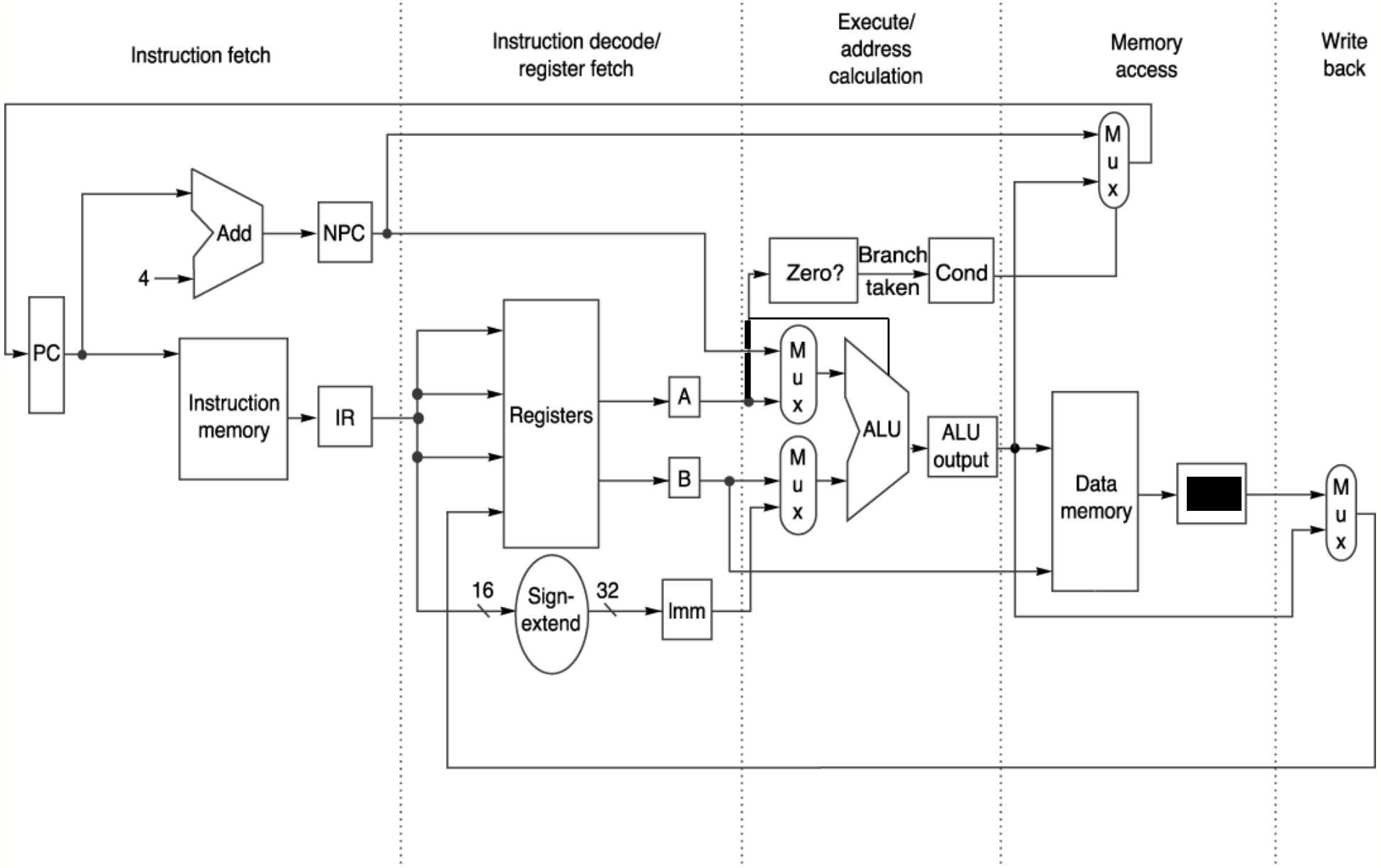
MIPS: Write Back

5. Write back Register (WB/REG):

- Reg[IR[20-16]] ‹- MDR
- The load operation is completed by moving the MDR content into the destination register specified in the instruction.
- During this phase, the Cu asserts RegWrite (WE) signal of register file, to allow writing of bits IR[20-16]

MIPS architecture datapath

Hennessy-Patterson, Fig. A.17:



MIPS instructions execution

- In this implementation, a branch takes 3 clock cycles, an arithmetic/logical I-type, R-type or store takes 4, and a load takes 5.
- Assuming (reasonable values, obtained by statistics on various RISC architectures), that the frequency of a branch is 12%, of a load 10%, the average number of clock cycles to execute an instruction (CPI) is 3,98 (check this !).
- Is there any gain with respect to a standard single cycle architecture?

The multicycle Control Unit

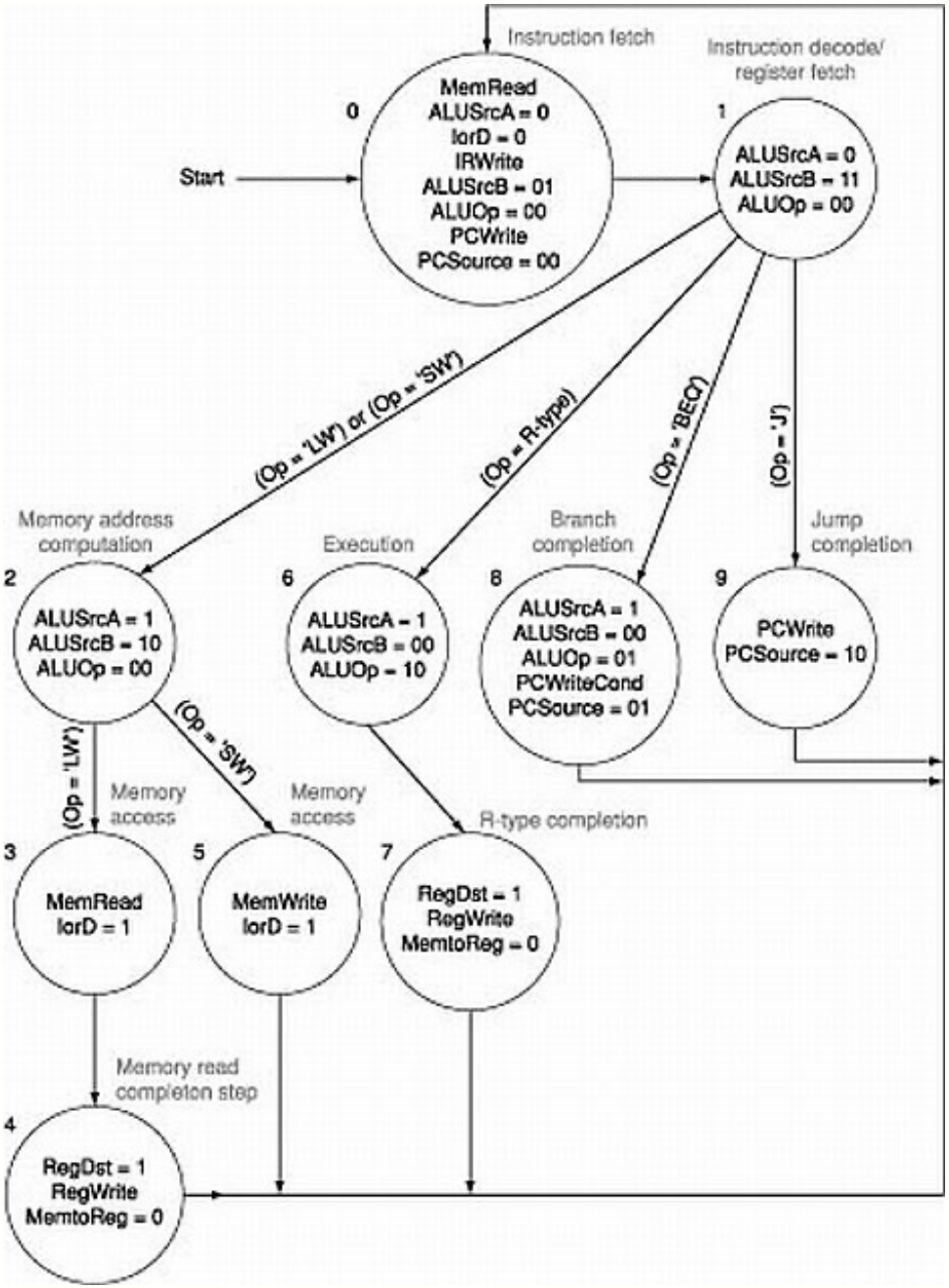
- How does the CU change in a multicycle version of MIPS ?
- For sure, it is more complex, because it has to specify which signals have to be asserted in each step, and which is the next step itself.
- Each step is described by a truth table whose inputs are:
  - the type of instruction being executed (the “op” field !)
  - the current step in the sequence describing the instruction execution

The multicycle Control Unit

- The output of the truth table at each step consists of:
  - the set of signals to be asserted
  - the next step in the sequence
- This is just an informal description of a **Moore machine**: a type of finite state automaton:
  - each state has an output (in this case, the signals to be asserted)that only depends on that state
  - the transition to the next state only depends on the current state and on the input (as in any finite state automaton)

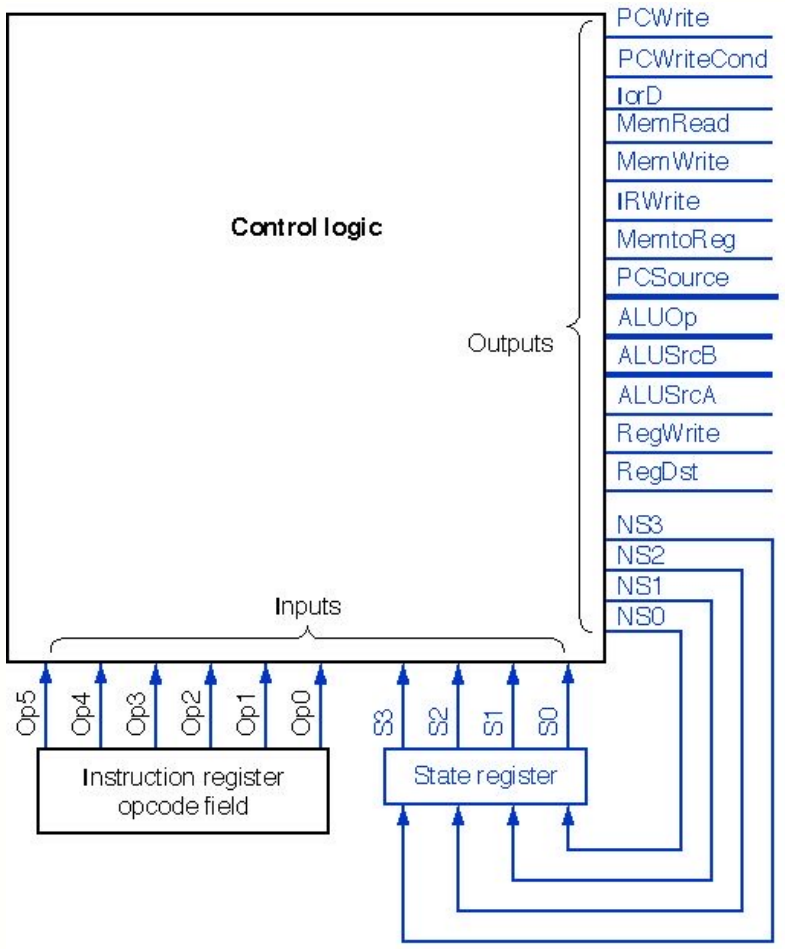
The multicycle Control Unit

- Here is the finite state machine describing the multicycle MIPS CU.
- The states match the steps described previously.
- As an instance, to execute a R-type instruction, it is neccessary to go through 4 states (0, 1, 6, 7); then back to the initial state (Patterson-Hennessy, fig. 5.38).



The multicycle Control Unit

- A finite state CU can be easily realized by combining a block of combinational logic with a register, that stores the current state of the machine.
- The next state depends only on the current one. (Patterson-Hennessy, fig. C.3.2).
- Question: why is the state register a 4-bit register? (go back to the previous slide).



Finite state machines and microprograms

- What is the difference between a datapath controlled by a finite state machine and another controlled by a microprogram? none, really.
- Microprogramming applies a symbolic representation of control based on instructions, to describe CU operation: microinstructions executed on a simple microprocessor.
- (Exercise: review Tnembaum's description of the *Instruction Fetch Unit*. What is being used?...)

Finite state machines and microprograms

- a microprogram is nothing more than a textual representation of a Moore finite state machine, with each microinstruction corrisponding to a state in the machine (so, what matches the "state register"?).
- So, why having a double representation (finite state automaton, and microprogram) for the CU operation ?
- It is "only" a matter of complexity of the CU, which actually depends on the number and richness of the ISA instructions.

Finite state machines and microprograms

- A real RISC architecture features many more instructions than we considered so far, and some take various clock cycles to be executed.
- The control in these architectures is definitely much more complex than the example we discussed, and consequently the associated automaton will be more complex too.
- Hardware description languages (“verilog”, “vhdl”) are used to described the control operations and to synthetise the CU as a finite state automaton.

Finite state machines and microprograms

- But, beyond a certain complexity, the automaton representation is no longer a viable means.
- As we will see, (possibly we already know this...), **CISC** architectures (what is a CISC architecture?...) usually feature hundreds of instructions, some of them really complex, since they combine arithmetic/logic operations with memory access.
- The number of steps to execute these instructions is very high, resulting in a finite state automaton with thousands of states and hundred of state sequences for many instructions

Finite state machines and microprograms

- Describing such a complex automaton, verifying its correctness, indentifing possible improvements to the datapath to optimize instructions execution, is an almost impossible task.
- Describing the control function with a microprogram simplifies the design and the analysis of the processor datapath, especially when this requires a very complex control.

Finite state machines and microprograms

- Microprogramming was the natural choice for architectures in the 60s and 70s; the trend was designing ISA with a very rich set of very complex and “powerful” instructions.
- This trend had solid practical reasons, to be analysed in the sequel.

Complex Instruction Set Computer

- The idea of describing the datapath control function through a microprogram dates back to 1953, when M. Wilkes proposed to use microcode for describing the execution of instructions.
- The basic idea was to introduce a further layer between the ISA level and instruction execution, decomposing instructions into finer steps, the microinstructions.
- At that time, the notion of finite state automaton was not yet formalized, it was defined later in the second half of the 50s.

Complex Instruction Set Computer

- This approach to control unit design lasted all th 60s and 70s long, and was well justified and matched to the technology available in those years.
- Computer Aided Design simply did not exist, so that describing and designing a CU with a complex finite state automaton was much more involved than using a microgrogramming approach.
- And in those years, the trend was designing architectures with a very rich and complex instruction set.

Complex Instruction Set Computer

- Very complex machine instructions were, at that time, a reasonable choice, with the technology available:

1. RAM access time was much longer than ROM access time, where the microprogram was stored, with ROM physically embedded in the CPU or located very close to it.

2. There existed no CPU with caches, the latter being introduced only at the beginning of the 80s.

Complex Instruction Set Computer

- It was quite reasonable that every access to the RAM to get the next instruction would bring to the CPU an instruction capable of deploying a “lot of work”.
- As an instance, IBM 370 sported an extremely powerful machine instruction:
  - MVC X Y L
    move a string of length L from address X to address Y
- Once brought to the CPU, the instruction could be executed by decomposing it into small steps (microinstructions)

Complex Instruction Set Computer

3. Compiler technology was still in its early phase; powerfull machine instructions made the task of the compiler much easier. The “semantics” of the instruction hided the complexity of the compilation phase !

4. RAM was expensive and limited, so powerfull instructions allowed to produces short executables.

Complex Instruction Set Computer

- Lastly, microprogramming in control made it simple to enlarging the ISA of CPU with new instructions:
- Simply, one could change the microprogram by adding new instructions in the microcode ROM.
- ROM had its own chip (it was not part of the CPU chip set), so it was reasonably easy to enlarge the ISA by changing the chip of the ROM only, no the whole CPU.

Complex Instruction Set Computer

- This flexibility in adding instructions motivated a many new proposal by CPU designers, since ROM capacity was farily large.
- More and more instructions were deviced, for new functions.
- The most blatant example is the 80x86 family, that has seen an enormous increase of its ISA in the years (and decades !).
- But this approach has its own drawbacks...

Complex Instruction Set Computer

- In the early 80s, processors such as VAX from DEC or IBM mainframes had over 300 different instructions and up to 200 modes for specifying operands (including type and addressing mode). A very versatile and complex ISA resulted in:
  - a large number of bits in instruction formats, to specify the type of operation, the number, type and position of the operands;
  - variable length instruction formats, because it was impossible to allocate a predefined, fixed number of bits for all instructions, which would waste a lot of space in most formats (in 80x86 ISA instruction length varies from 1 to 17 bytes).

Complex Instruction Set Computer

- Complex and irregular (in their format) instructions require more time for decoding and for starting the actual execution (EX phase in MIPS).
- To begin with, operands cannot be retrieved (from the register file, as an instance) until the instruction type is known, because the addressing bits in the instruction vary in position according to instruction type.
- Complex instructions require a more advanced CU, made up with more logic gates, so that outputs from the CU to the datpath control points take longer to stabilize, once input are stable.

Complex Instruction Set Computer

- CISC ISA hides an even bigger problem: a versatile instruction format resulted in operands coming not only from registers::

  ADD R1, R2, R3 // R1⟵R2 + R3

- but also from registers AND memory locations:

  (an exam from VAX instruction):

  ADDL3 42 (R1), 56 (R2), 0 (R3)

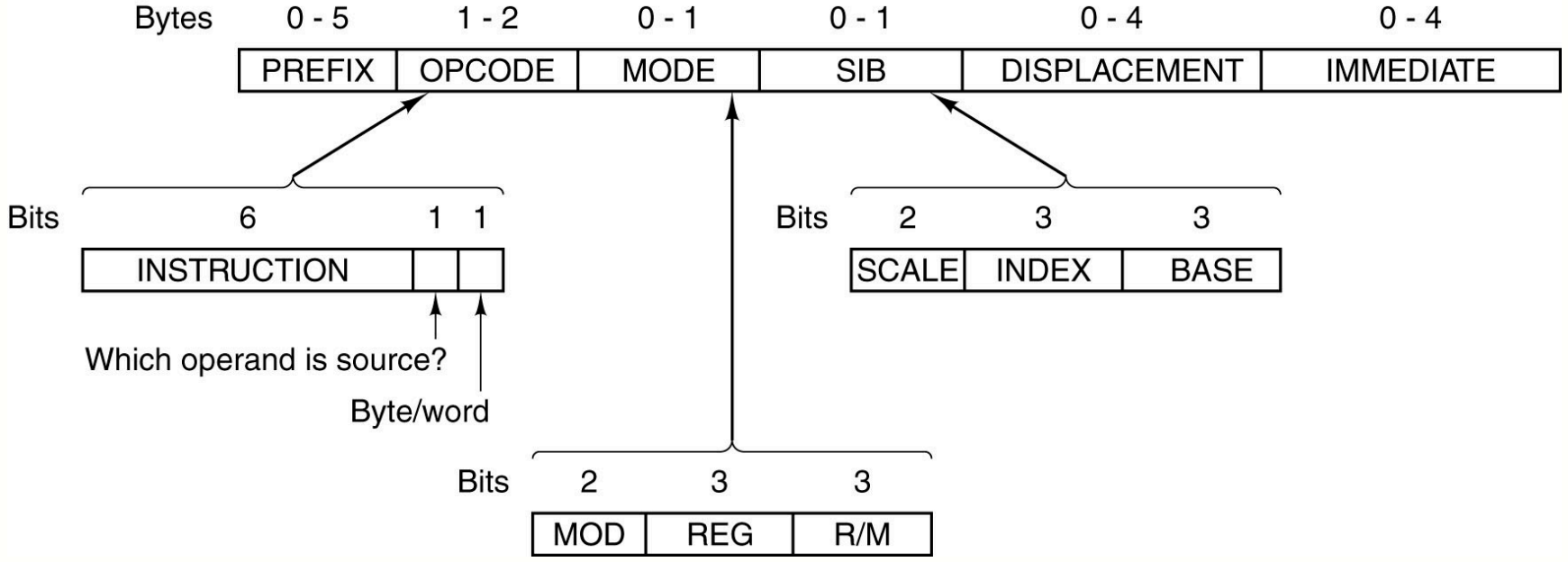
  (fetch a word from address 56 + [R2], a word from address 0 + [R3], add the two values and store the result at address 42 + [R1])

Complex Instruction Set Computer

- Indeed, instructions using explicitly memory locations yielded in a very compact code
  - as an instance, it is not necessary to produce code to bring operands to CPU registers, with was accompliced through the microcode
- **However, strong bottlenecks would arise easily in main memory access (itself much slower than the CPU)**
  - potentially, each CISC instruction could use one o more operands to be fetched from main memory.

An example: Pentium 4 ISA:

- This ISA is the outcome of the evolution of the 80c86 architecture 80x86 across multiple versions, with backward compatibility as a must.
- It is complex and irregular, with up to 6 fields, each of them variable in length; 5 of the 6 are optional.
- One of the operands (not both ! at least) in (almost) every instruction can be in memory (Tanenbaum, Fig. 5-14)



Pentium 4 ISA

- A *small section* of the *most common* integer instructions in Pentium 4 (Tanenbaum, Fig. 5-34)

Moves

MOV DST,SRC	Move SRC to DST
PUSH SRC	Push SRC onto the stack
POP DST	Pop a word from the stack to DST
XCHG DS1,DS2	Exchange DS1 and DS2
LEA DST,SRC	Load effective addr of SRC into DST
CMOVcc DST,SRC	Conditional move

Arithmetic

ADD DST,SRC	Add SRC to DST
SUB DST,SRC	Subtract SRC from DST
MUL SRC	Multiply EAX by SRC (unsigned)
IMUL SRC	Multiply EAX by SRC (signed)
DIV SRC	Divide EDX:EAX by SRC (unsigned)
IDIV SRC	Divide EDX:EAX by SRC (signed)
ADC DST,SRC	Add SRC to DST, then add carry bit
SBB DST,SRC	Subtract SRC & carry from DST
INC DST	Add 1 to DST
DEC DST	Subtract 1 from DST
NEG DST	Negate DST (subtract it from 0)

Binary coded decimal

DAA	Decimal adjust
DAS	Decimal adjust for subtraction
AAA	ASCII adjust for addition
AAS	ASCII adjust for subtraction
AAM	ASCII adjust for multiplication
AAD	ASCII adjust for division

Boolean

AND DST,SRC	Boolean AND SRC into DST
OR DST,SRC	Boolean OR SRC into DST
XOR DST,SRC	Boolean Exclusive OR SRC to DST
NOT DST	Replace DST with 1's complement

Shift/rotate

SAL/SAR DST,#	Shift DST left/right # bits
SHL/SHR DST,#	Logical shift DST left/right # bits
ROL/ROR DST,#	Rotate DST left/right # bits
RCL/RCR DST,#	Rotate DST through carry # bits

Test/compare

TEST SRC1,SRC2	Boolean AND operands, set flags
CMP SRC1,SRC2	Set flags based on SRC1 - SRC2

Condition codes

STC	Set carry bit in EFLAGS register
CLC	Clear carry bit in EFLAGS register
CMC	Complement carry bit in EFLAGS
STD	Set direction bit in EFLAGS register
CLD	Clear direction bit in EFLAGS reg
STI	Set interrupt bit in EFLAGS register
CLI	Clear interrupt bit in EFLAGS reg
PUSHFD	Push EFLAGS register onto stack
POPFD	Pop EFLAGS register from stack
LAHF	Load AH from EFLAGS register
SAHF	Store AH in EFLAGS register

Miscellaneous

SWAP DST	Change endianness of DST
CWQ	Extend EAX to EDX:EAX for division
CWDE	Extend 16-bit number in AX to EAX
ENTER SIZE,LV	Create stack frame with SIZE bytes
LEAVE	Undo stack frame built by ENTER
NOP	No operation
HLT	Halt
IN AL,PORT	Input a byte from PORT to AL
OUT PORT,AL	Output a byte from AL to PORT
WAIT	Wait for an interrupt

SRC = source
DST = destination
# = shift/rotate count
LV = # locals

Transfer of control

JMP ADDR	Jump to ADDR
Jxx ADDR	Conditional jumps based on flags
CALL ADDR	Call procedure at ADDR
RET	Return from procedure
IRET	Return from interrupt
LOOPxx	Loop until condition met
INT n	Initiate a software interrupt
INTO	Interrupt if overflow bit is set

Strings

LODS	Load string
STOS	Store string
MOVS	Move string
CMPS	Compare two strings
SCAS	Scan Strings

Reduced Instruction Set Computer

- With CISC processor production still flourishing, at the begin of 80s new machines are being designed according to a completely new design and conception.
- In1980, in Berkeley, D. Patterson and C. Sequin design a CPU whose Cu is not described by a microprogram, and cast the name **RISC** (actually, CISC as well, as a counterpart to RISC)
- Their project will result in the **SPARC** (**S**calable **P**rocessor **ARC**hitecture) systems by SUN.

Reduced Instruction Set Computer

- Almost at the same time, in 1981, in Stanford, J. Hennessy designs a similar architecture, capable of effectively deploying pipelining, and calls it **MIPS**: **M**icroprocessor *without* **I**nterlocked **P**ipeline **S**tages (with a nice word trick in the acronym …) later used in various commercial processors.
- To a large measure, it is the MIPS we are considering.
- Both architectures build on CRAY and on IBM 801 (a research prototype) from the 60s and 70s.

Reduced Instruction Set Computer

- In mid 70s, a few researchers (among which J. Cocke from IBM) had shown that programmers (and compilers) used only a small fraction of the addressing modes available in CISC ISA of that period.
- A. Tanenbaum had shown through statistics the almost all instructions used small, immediate values, that could be represented with 13 bits, while instructions formats usually allowed 16 o 32 bits to store the “immediate”. Most bits went unused most of the times.

Reduced Instruction Set Computer

- Stated another way, most programs spend most of the time executing simple instructions.
- It is therefore useless having a large set of long and complex instructions (moreover, of variable length), that:
  - took a lot of time to load from RAM (their length in bytes cused multiple memory accesses)
  - required a lot of work to be decoded.

Reduced Instruction Set Computer

- Which is the driving idea in RISC machines?

1. The CPU executes a limited number of simple instructions that can be (ultimately) executed in a single clock cycle.

- Simple instructions demand a smaller datapath, a simple CU, with short time to apply control (the time necessary to produce datapath control signals once *op* and *funct* fields are input to the CU).

- For these reasons, it is possible to used a short clock cycle (decomposing the execution in more clock cycle is a winning approach, nethertheless).

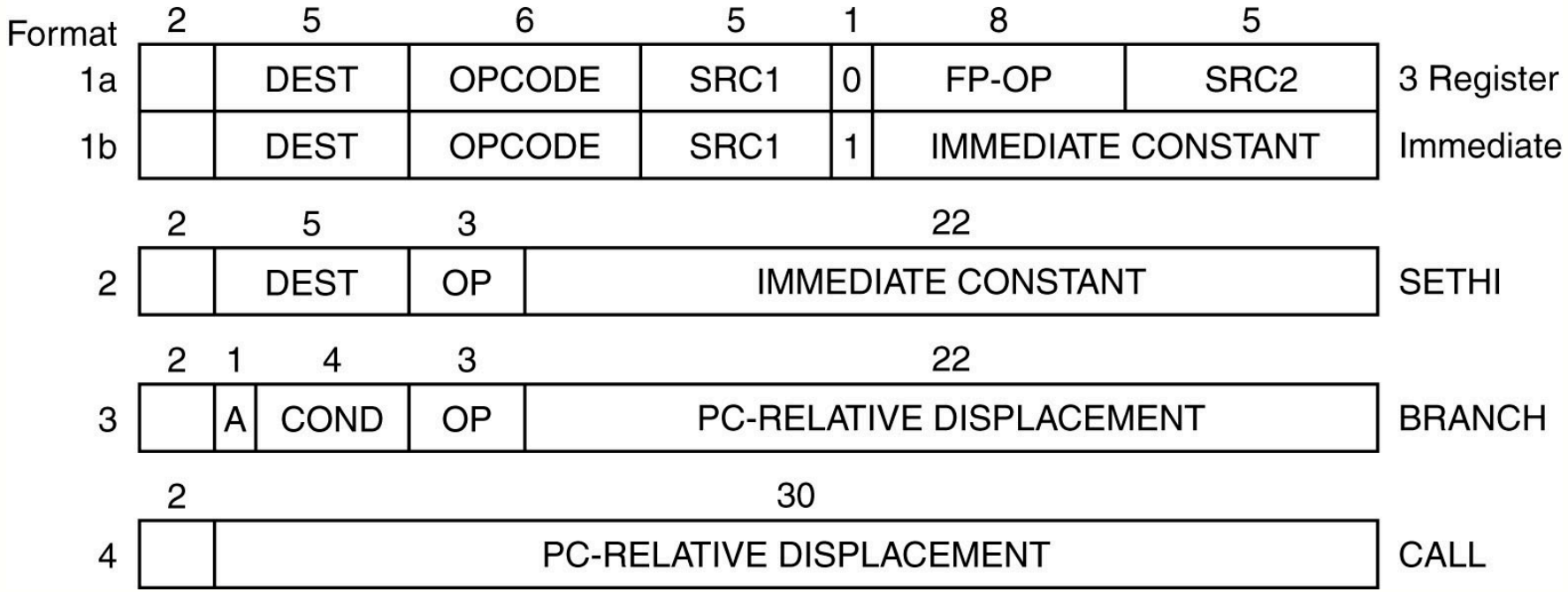
Reduced Instruction Set Computer

2. Instructions mainly access operands in registers (within the CPU) and store the result in a register too.

3. Access to RAM is limited as much as possible (even if caches are available), and is restricted only to special instructions, LOAD and STORE.

An example: SPARC ISA

- All instructions take 32 bits.
- Typically, the operands are three registers
- With time, instruction format has developed with vaious SPARC versions, but it has retained a fixed length and a fixed and regular field decomposition (Tanenbaum, fig. 5-15)



UltraSPARC III ISA

- *All* integer instructions in UltraSPARC III. Note the unique instructions that access main memory (Tanenbaum, Fig. 5-35)

Loads

LDSB ADDR,DST	Load signed byte (8 bits)
LDUB ADDR,DST	Load unsigned byte (8 bits)
LDSH ADDR,DST	Load signed halfword (16 bits)
LDUH ADDR,DST	Load unsigned halfword (16)
LDSW ADDR,DST	Load signed word (32 bits)
LDUW ADDR,DST	Load unsigned word (32 bits)
LDX ADDR,DST	Load extended (64-bits)

Stores

STB SRC,ADDR	Store byte (8 bits)
STH SRC,ADDR	Store halfword (16 bits)
STW SRC,ADDR	Store word (32 bits)
STX SRC,ADDR	Store extended (64 bits)

Shifts/rotates

SLL R1,S2,DST	Shift left logical (32 bits)
SLLX R1,S2,DST	Shift left logical extended (64)
SRL R1,S2,DST	Shift right logical (32 bits)
SRLX R1,S2,DST	Shift right logical extended (64)
SRA R1,S2,DST	Shift right arithmetic (32 bits)
SRAX R1,S2,DST	Shift right arithmetic ext. (64)

Arithmetic

ADD R1,S2,DST	Add
ADDCC "	Add and set icc
ADDC "	Add with carry
ADDCCC "	Add with carry and set icc
SUB R1,S2,DST	Subtract
SUBCC "	Subtract and set icc
SUBC "	Subtract with carry
SUBCCC "	Subtract with carry and set icc
MULX R1,S2,DST	Multiply
SDIVX R1,S2,DST	Signed divide
UDIVX R1,S2,DST	Unsigned divide
TADCC R1,S2,DST	Tagged add

Boolean

AND R1,S2,DST	Boolean AND
ANDCC "	Boolean AND and set icc
ANDN "	Boolean NAND
ANDNCC "	Boolean NAND and set icc
OR R1,S2,DST	Boolean OR
ORCC "	Boolean OR and set icc
ORN "	Boolean NOR
ORNCC "	Boolean NOR and set icc
XOR R1,S2,DST	Boolean XOR
XORCC "	Boolean XOR and set icc
XNOR "	Boolean EXCLUSIVE NOR
XNORCC "	Boolean EXCL. NOR and set icc

Miscellaneous

SETHI CON,DST	Set bits 10 to 31
MOVcc CC,S2,DST	Move on condition
MOVr R1,S2,DST	Move on register
NOP	No operation
POPC S1,DST	Population count
RDCCR V,DST	Read condition code register
WRCCR R1,S2,V	Write condition code register
RDPC V,DST	Read program counter

Reduced Instruction Set Computer

- Specifically, the good design rules for a modern CPU are the following:

- **No sofisticated microcode**
  - A simple CU commutes in a shorter time, and can use a shorter clock cycle (higher frequency).
  - RISC instructions are simpler than CISC ones, and to produce the result obtained with a single CISC instrutions, it can be necessary to use 4 or 5 RISC ones. This is no longer a problem, with large inexpensive RAM memory modules.

Reduced Instruction Set Computer

- **Define instructions easily decoded**
  - To be executed, an instruction must be decoded, so that it is possibile to know the operation to be carried out, the resources required (which functional units, which are the operands, where the result will be stored).
  - Fixed length instructions, with few operands and few addressing modes make decoding quicker.

Reduced Instruction Set Computer

- **RAM memory must be addressed only through LOAD and STORE instructions**
  - Since RAM is much slower than CPU in modern processors, (the gap was smaller in the 60s and 70s, but no caches were available), the least it is used, the better.
  - As far as possible, all instructions must work on registers, using RAM only when operands must be fetched from memory, o stored back.
  - Off course, using caches alleviates RAM slow access, but RAM use must be kept to a minimum.

Reduced Instruction Set Computer

- **Having a lot of general-purpose registers**
  - Since RAM is slow, having a lot of registers allows to keep intermediate results within the CPU, without having to resort to RAM.
  - Indeed, when all registers are in use, the result of a computation must be stored in RAM, with a waste in time.
  - In all modern architectures, a special effort is placed on having as large a number of general purpose registers as possible.

Reduced Instruction Set Computer

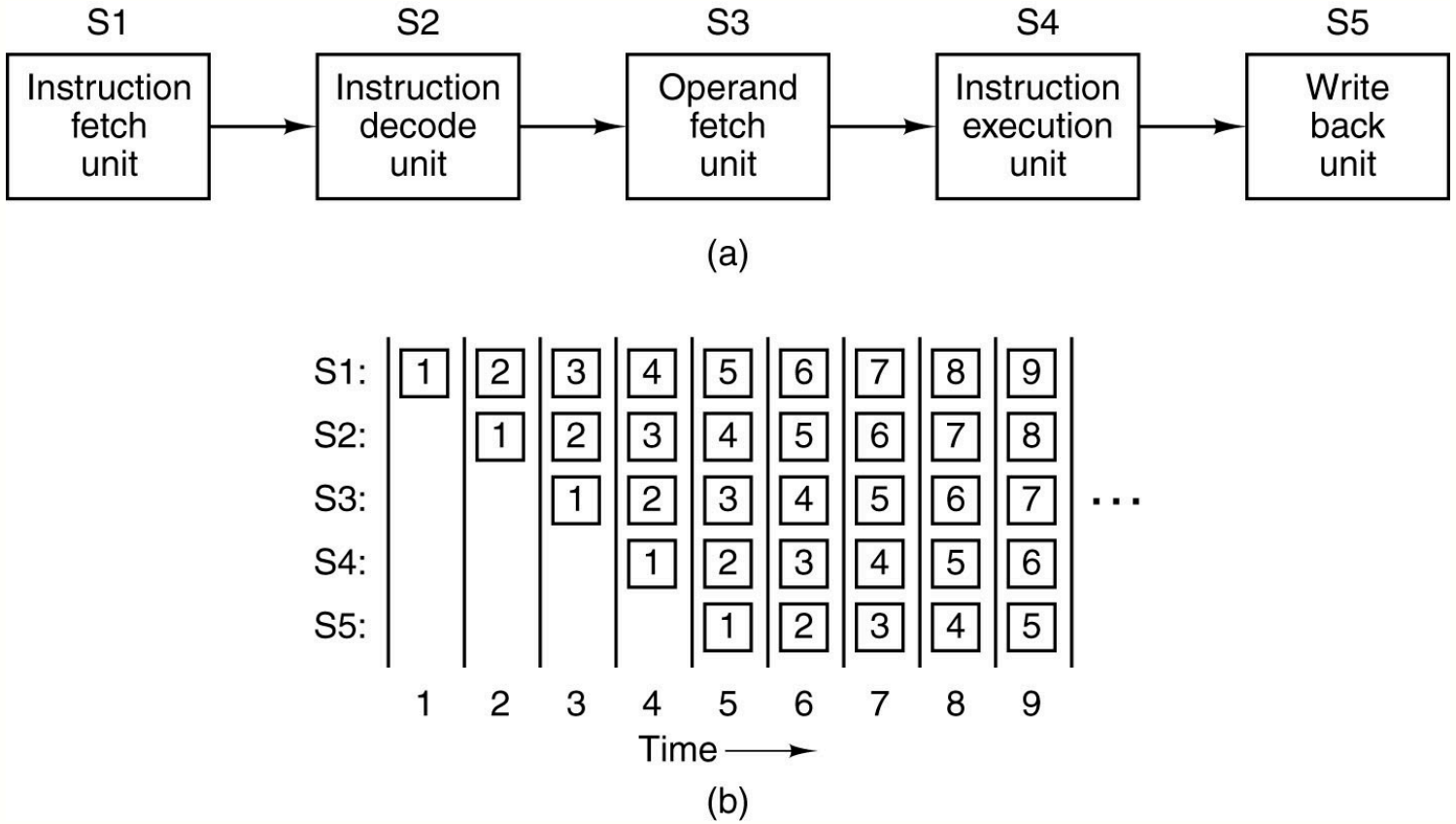
- Let us illustrate these concepts with an example using the VAX CISC instruction just considered:
  - ADDL3 42 (R1), 56 (R2), 0 (R3)
- in a RISC architecture, it corresponds to the following sequence:
  - LD R4, 56 (R2)
  - LD R5, 0 (R3)
  - ADD R6, R4, R5
  - SD R6, 42 (R1)
- So, a longer program section, with more registers, and RAM used only with load and store instruction.

Reduced Instruction Set Computer

- Finally: **exploiting pipelining as much as possible**
  - If the architecture is well designed, the execution of the instruction can be naturally split into many separate phases.
  - In principle, *each phase of a given instruction can be executed in parallel to a different phase of another instruction*: this is pipelining

Reduced Instruction Set Computer

- **exploiting pipelining as much as possible**
  (Tanenbaum, Fig. 2-4 – adapted to MIPS)



Reduced Instruction Set Computer

- **exploiting pipelining as much as possible**
  - If a pipeline is split into 5 phases, and a clock cycle (say, 2 ns) is required to go from one phase to the other, an instruction is executed in 10 nanoseconds. Processors MIPS = 100.
  - But if pipelining is exploited at the maximum level, every 2 ns a new instruction can be launched, and every 2 ns an instruction completes execution, so the "virtual" MPIS of the processor raises to 500.
  - Actually, this is only the theoretical case, since there is a chance that the pipeline gets "stalled" (we'll cover this in detail later); also a pipelined architecture is slightly more complex (and thus slower) than a non pipelined one.

Reduced Instruction Set Computer

- **exploiting pipelining as much as possible**
  - Pipelining is a technique that best suites RISC architectures (it is a complex technique, yet). The processor name MIPS stands for **Microprocessor *without* Interlocked Pipelines Stages**, just to highlight an architecture with few dependencies within instructions, so as to exploit as much as possible the parallelism inherently embedded in the instruction (ILP).

  - In the 80x86 family, pipelining has shown up only in 486, and two pipelines have been used later in Pentium (with Pentium, II Intel reverted to a single pipeline, but with a *multiscalar* architecture)

Reduced Instruction Set Computer

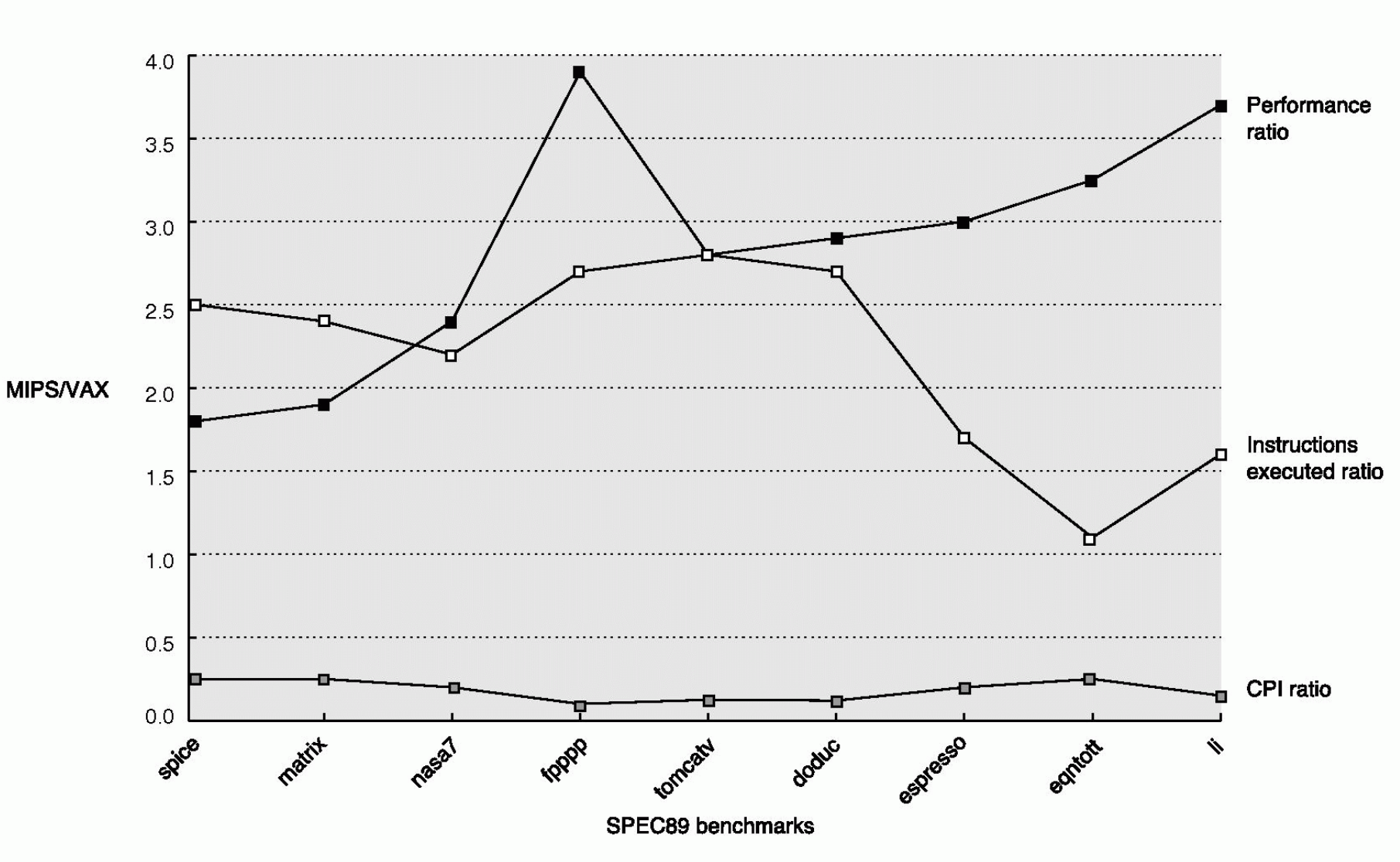
- What has been the follow up in the RISC vs CISC theme?
- In a paper published in1980, Clark and Strecker (the designers of the VAX architecture) are very sceptical that a RISC architecture can ever compete with CISC architectures.
- In 1986 some companies (HP, IBM) start producing RISC processors.
- In 1987 Sun Microsystems begins to produce a processor with a SPARC architecture, based on the RISC design originated from Patterson in Berkley.

Reduced Instruction Set Computer

- In 1990 IBM announces its first superscalar RISC architecture (RS6000), based on the IBM 801 concept (never gone to production)
- In the late 80s, even VAX designers assess the strength of the RISC architecture, by comparing the performances of the two most powerful processors of that time:VAX 8700 and MIPS M2000.
- The MIPS machine was found to be roughly 3 times quicker than VAX in various benchmarks.
- In early 90s, Digital cancels VAX production and starts producing processors based on the Alpha architecture, very similar to MIPS.

Reduced Instruction Set Computer

- On the average, MIPS executes twice as many instructions as VAX, but VAX CPI is 6 times larger than MIPS' (Hennessy – Patterson, Fig. 2.41)



CISC vs RISC

- After 25 years, who has won the CISC/RISC war?
- Obviously RISC, since all companies in the computer/CPU market have switched to RISC based production.
- Moreover, in 2000, the embedded 32-bit CPU market (deployed in smart and video cameras and in other appliances) is 90% RISC
- A single CISC architecture has survived the war well, and exceptionally well indeed ….

CISC vs RISC

- The Intel 80x86 family (from 8086 until Pentium 4) and the subsequent multicore family still have a large share in the desktop/laptop market. Why?
  - The main reason is backward compatibility: modern Intel processors still execute code designed for 8086, the first 16-bit single chip microprocessor (1978). On another side, IBM chose 8086 (and later 80286) for its PC line of products.
  - Billions of dollars have been spent to develop software for the 80x86 architecture, and backward compatibility guarantees that all this software is still in use.

CISC vs RISC

- Furthermore, Intel designers have succeded in deploying RISC design principles in their new processors.
- A single pipeline (a “natural” choice for RISC architectures) was first introduced in 486. The “586” has a double pipeline; Pentium II switches to a *superscalar* structure, an idea that dates back by 40 years, at the time of CDC 6600, a forerunner of RISC architettures.
- Actually, since 486 all Intel CPUs host a RISC “core” that allows to execute sinple and frequently issued instructions in very few clock cycles (more complex instructions are interpreted according to the standard CISC microcode mode).

CISC vs RISC

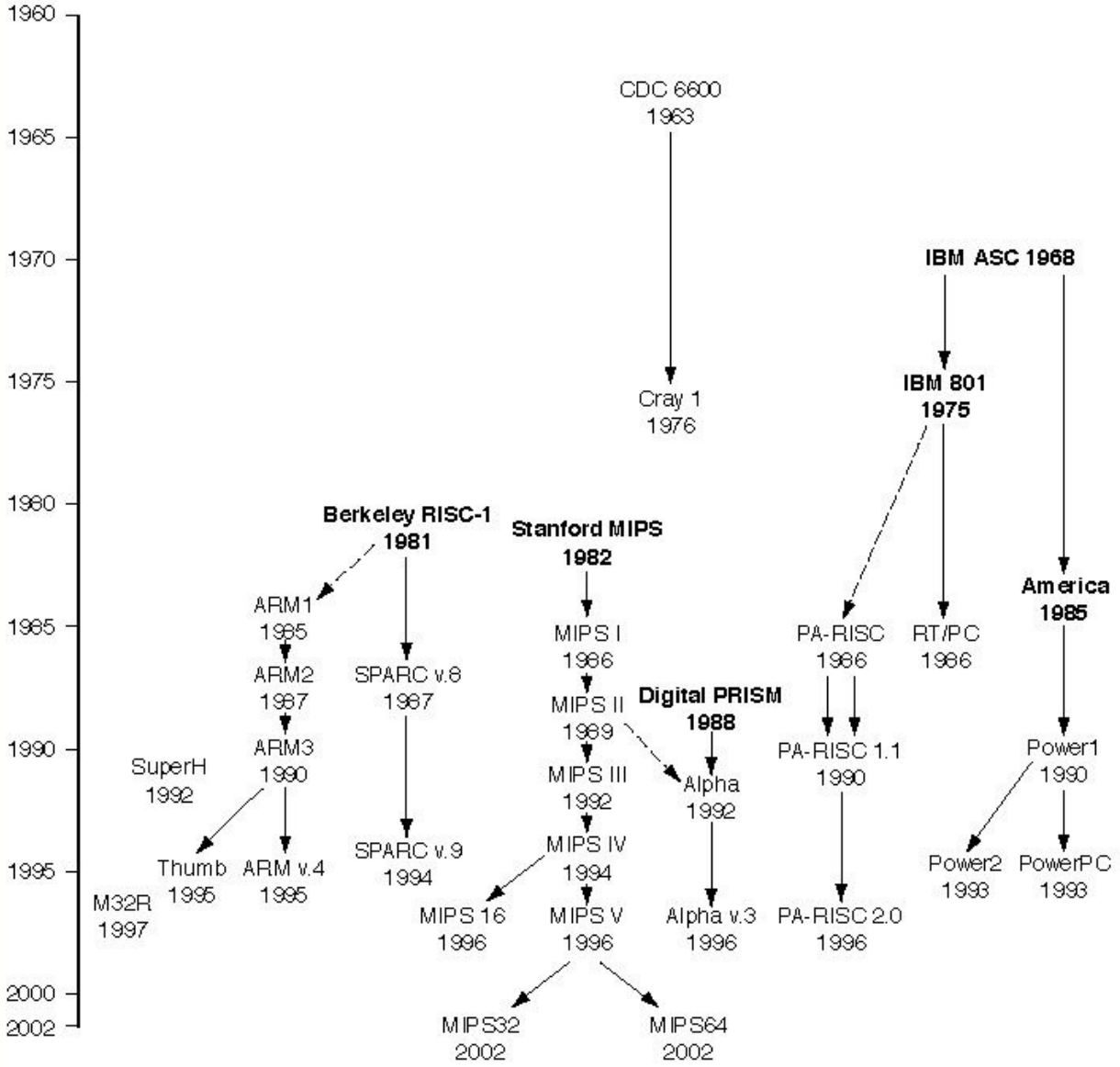
- Intel launched (with uncertain results) a full RISC project, **ITANIUM 2,** a complete departure from the processor series 80x86 / core duo / i7-i5-i3.
- It is a 64-bit RISC architecture, which we'll cover in the section on static ILP.

CISC vs RISC

- Currently, the distinction CISC – RISC has little meaning.
- On one side, the “commonly adopted” design principles are those of RISC architectures.
- On another side, modern architectures are much more complex and sofisticated than those typical 30 years ago, and they actually “look like” old CISC architectures, at a closer look!

RISC architectures genealogy tree

- Dotted lines show families of embedded processors, processors drawn in bold are research machines never gone to production. (Patterson-Hennessy, fig. D.17.2)



Measuring performance

- Typical performance metrics:
  - Response time
  - Throughput

- Speedup of X relative to Y
  - Execution time$_Y$ / Execution time$_X$

- Execution time
  - Wall clock time: includes all system overheads
  - CPU time: only computation time

- Benchmarks
  - Kernels (e.g. matrix multiply)
  - Toy programs (e.g. sorting)
  - Synthetic benchmarks (e.g. Dhrystone)
  - Benchmark suites (e.g. SPEC06fp, TPC-C)

Quantative principles

- Improving performance:

  – Taking advantage of *parallelism*

    At system level: OS scheduling threads

    At the *processor* level: many-core, GPUs

    At the *core* level: pipelining

    At *digital design* level: SIMD CPUs, associative caches

  – Locality

    Reuse of data and instructions accessed “recently”

  – The Common case

    Optimize resources by favoring the most frequent case

Measuring performance

Amdahl's Law

- Focus on the common case
- Speedup that can be obtained using a specific enhancement

$$Speedup = \frac{Execution\ time\ without\ enhancement}{Execution\ time\ with\ enhancement\ when\ possible}$$

$$\text{Execution time}_{new} = \text{Execution time}_{old} \times \left((1 - \text{Fraction}_{enhanced}) + \frac{\text{Fraction}_{enhanced}}{\text{Speedup}_{enhanced}} \right)$$

$$\text{Speedup}_{overall} = \frac{\text{Execution time}_{old}}{\text{Execution time}_{new}} = \frac{1}{(1 - \text{Fraction}_{enhanced}) + \frac{\text{Fraction}_{enhanced}}{\text{Speedup}_{enhanced}}}$$

Measuring performance

Amdahl's Law – example 1

•A processor has to be enhanced for Web processing: the enhanced version is 10 times faster in Web application. Assuming the old processor is idle 60% of time for I/O operations, and carries out computations for a fraction 0.4 of time = (40%), what is the overall speed up gained by using the enhancement ?

$$\text{Speedup}_{\text{overall}} = \frac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

•$\text{Fraction}_{\text{enhanced}}$=0.4; $\text{Speedup}_{\text{enhanced}}$=10

$$Speedup_{overall} = \frac{1}{0.6 + \frac{0.4}{10}} = \frac{1}{0.64} = 1.56$$

Measuring performance

Amdahl's Law – example 2

•Graphics processors use consistently FP operations; FP square root (FPSQRT) takes 20% of execution time in a *given relevant benchmark*. Two alternatives (that are supposed to cost the same in silicon):

•1) Better hardware for FPSQRT allows for 10 times speedup in this op.

•2) All FP operations are made 1.6 faster (FP are 50% of overall workload)

$$Speedup_{FPSQRT} = \frac{1}{(1-0.2)+\frac{0.2}{10}} = \frac{1}{0.82} = 1.22$$

$$Speedup_{FP} = \frac{1}{(1-0.5)+\frac{0.5}{1.6}} = \frac{1}{0.8125} = 1.23$$

Measuring performance

Amdahl's Law in Parallel Programming

- The law assumes a fixed amount of work ("problem size fixed")
- Ex: an image of size NxN pixels, the task FFT
- The *enhancement* is the use of more processors *n*; the uniprocessor case n=1 is the baseline
- The speed up is limited by the amount *p* of work that can be actually done in parallel

$$Speed\text{-}up = \frac{n}{n + p\,(1\text{-}n)}$$

$p = 0$; (all *serial*) $Speed\text{-}up = 1$

$p = 1$; (all *parallel*) $Speed\text{-}up = n$

- If the assumption of fixed amout is relaxed, another law describes the performace, the Gustavson's Law (to be discussed later)

A useful figure of merit:

- **CPI = Clock cycles Per Instruction**
- That is, the number of clock cycles to complete an instruction.
- This quantity shows the speed at which a CPU is capable of "completing" the instructions it is executing.
- In any ISA, different instructions require a different number of clock cycles to be completed. In a generic CPU, how can we compute its CPI, and which value can we expect for it?
- What would be the "ideal" CPI for a CPU?

A useful figure of merit

The processor performance equation

- **CPU time** = CPU_clock_cycles_for_a_program x Clock_cycle_time
- **IC** = *instruction count (instruction pathlength)*
- **CPI** = CPU_clock_cycles_for_a_program / IC (**C**lock cycles **P**er **I**nstruction)
- **CPU time= IC** x **CPI** x Clock_cycle_time

$$\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}} = \frac{\text{Seconds}}{\text{Program}} = \text{CPU time}$$

IC ⟶ Instruction set architecture and compiler technology

CPI ⟶ organization and instruction set architecture

Clock_cycle_time ⟶ hardware technology and organization

A useful figure of merit

The processor performance equation

- If an architecture has different PCI for a set of *i* classes of instructions (which is typical of CISC ISA)

$$\text{CPU clock cycles} = \sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i$$

$$\text{CPU time} = \left(\sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i \right) \times \text{Clock cycle time}$$

$$\text{CPI} = \frac{\sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i}{\text{Instruction count}} = \sum_{i=1}^{n} \frac{\text{IC}_i}{\text{Instruction count}} \times \text{CPI}_i$$

A useful figure of merit

The processor performance equation

- Example: 4 classes (n=4)

			IC_i / IC
• i:1	Integer	CPI=5	50%
• i:2	Float add/sub	CPI=8	30%
• i:3	Float mult	CPI=11	15%
• i:4	Fload div	CPI=24	5%

$$\text{CPI} = \frac{\sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i}{\text{Instruction count}} = \sum_{i=1}^{n} \frac{\text{IC}_i}{\text{Instruction count}} \times \text{CPI}_i$$

$$= 5*0,5+8*0,3+11*0,15+24*0,05= 7,75$$

A useful figure of merit

CPI in pipelined CPUs

- L: number of pipeline stages
- IC: instruction count in the algorithm
- ***Ideal case***: no pipeline stalls

$$CPU\ time = (L-1) + IC$$

$$CPI_{IDEAL} = \frac{(L-1)+IC}{IC} = 1 + \frac{L-1}{IC} \approx 1$$

- ***Real case***: pipeline stalls

$$CPI_{PIPELINE} = CPI_{IDEAL} + AverageClockCyclesLostInStalls$$

A useful figure of merit

Speedup in pipelined CPUs

- L: number of pipeline stages

$$Speedup = \frac{\text{CPUtime}_{\text{nopipe}}}{\text{CPUtime}_{\text{pipe}}} = \frac{\text{IC x CPI}_{\text{nopipe}} \text{ x Clock\_cycle\_time}_{\text{nopipe}}}{\text{IC x CPI}_{\text{pipe}} \text{ x Clock\_cycle\_time}_{\text{pipe}}}$$

- Assumption: same Clock_cycle_time

$$Speedup = \frac{\text{CPI}_{\text{nopipe}}}{\text{CPI}_{\text{pipe}}} = \frac{\text{L}}{\text{1+AverageClockLostStalls}}$$