CO20-320241

Computer Architecture and Programming Languages

CAPL

Lecture 20 & 21

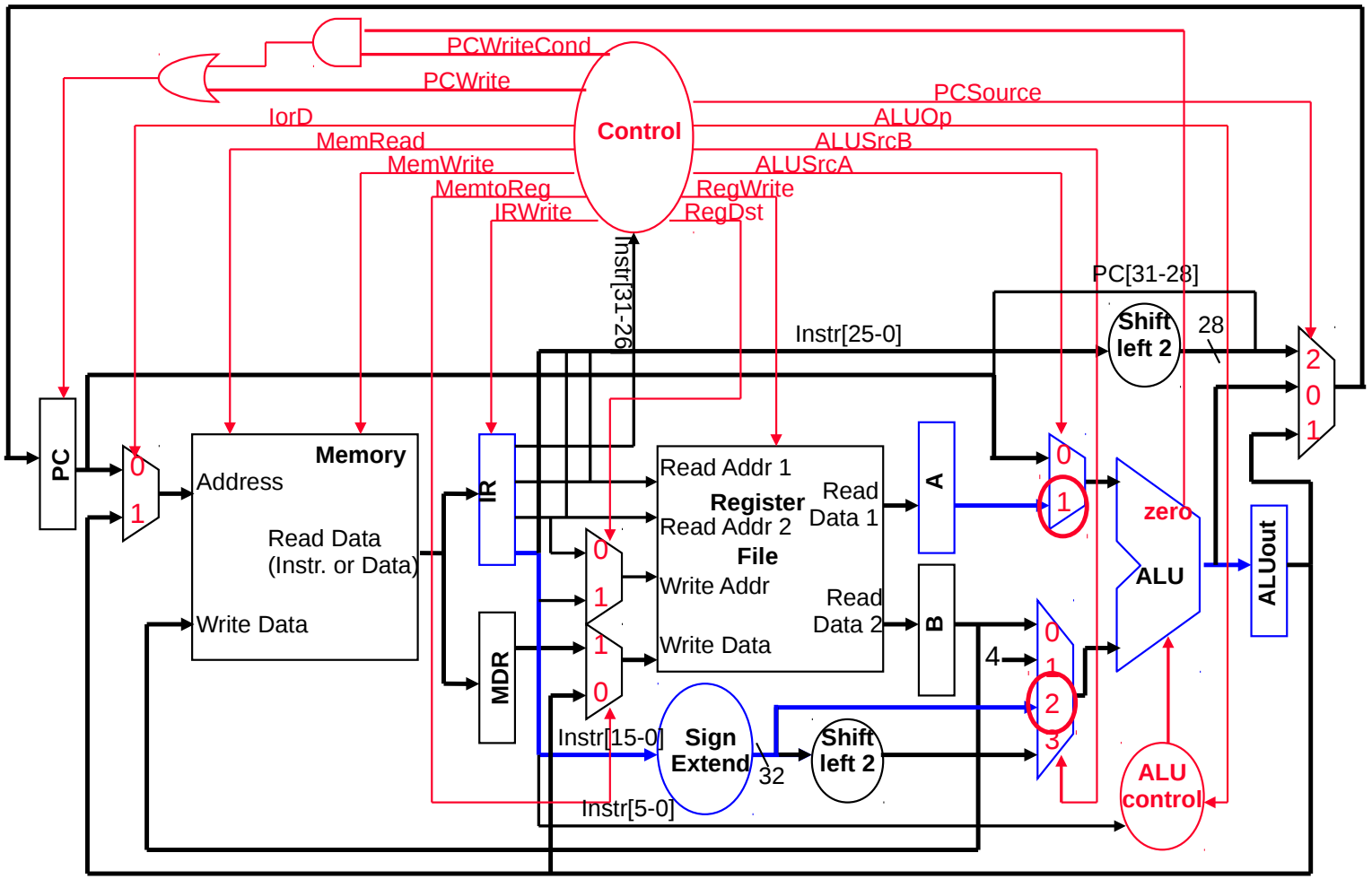
Dr. Kinga Lipskoch

Fall 2019

Step 3: Execution, Memory Address Computation or Branch Completion

- First cycle where step depends on the instruction
- Selection performed by interpretation of the op + function field of the instruction
- Memory reference
  - calculate address
    `ALUOut <= A + sign-extend(IR[15:0])`
  - Set ALUSrcA = 1 get operand from A
  - Set ALUSrcB = 10 get operand from sign extension unit
  - Set ALUOp = 00 add

Step 3: Memory Reference



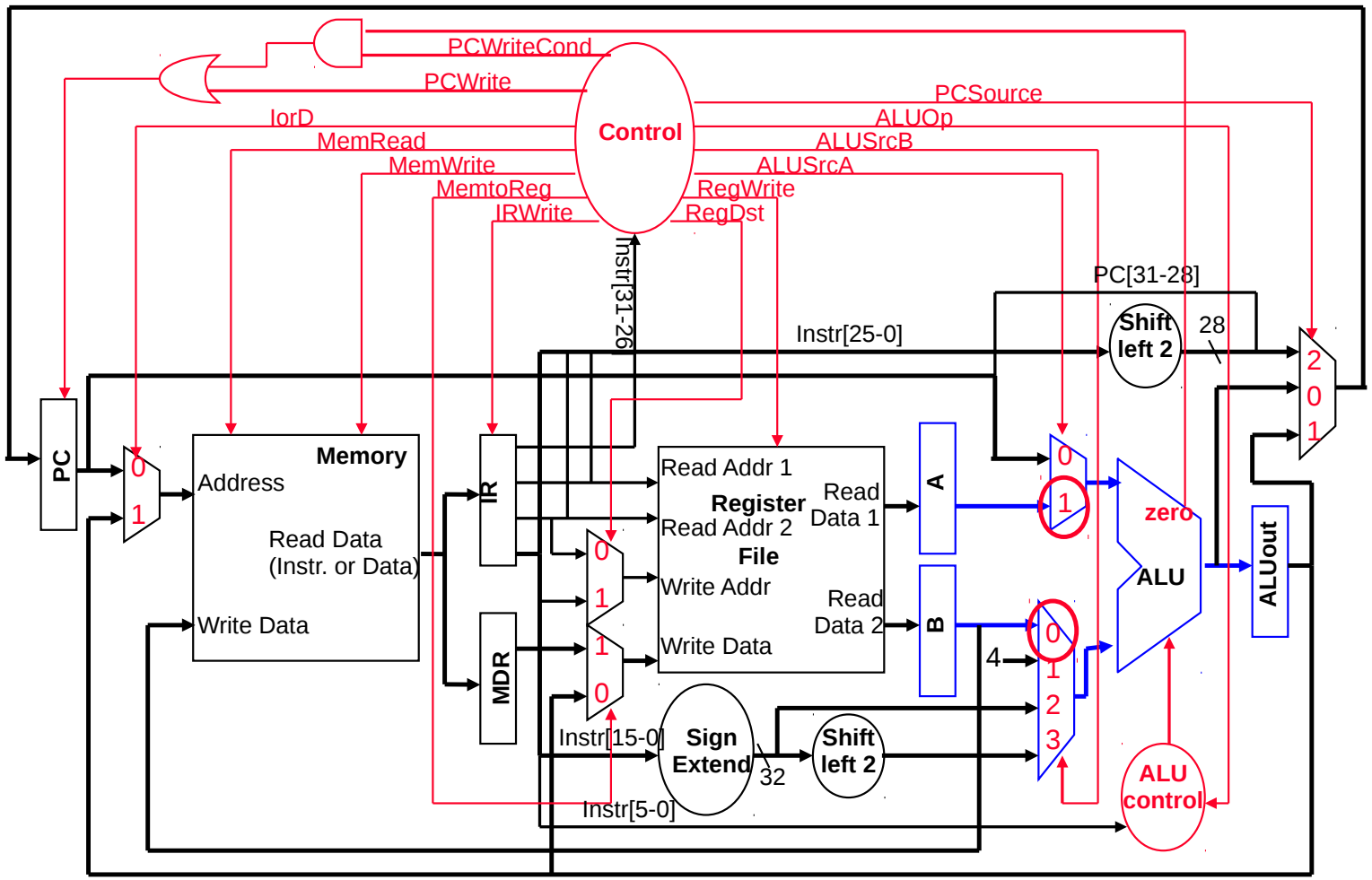
Step 3: Execution, Memory Address Computation or Branch Completion

Arithmetic-logical instruction (R-type):

```
ALUOut = A op B
```

- Set ALUSrcA = 1 get operand from A
- Set ALUSrcB = 00 get operand from B
- Set ALUOp = 10 code from IR

Step 3: Arithmetic-Logical Instruction



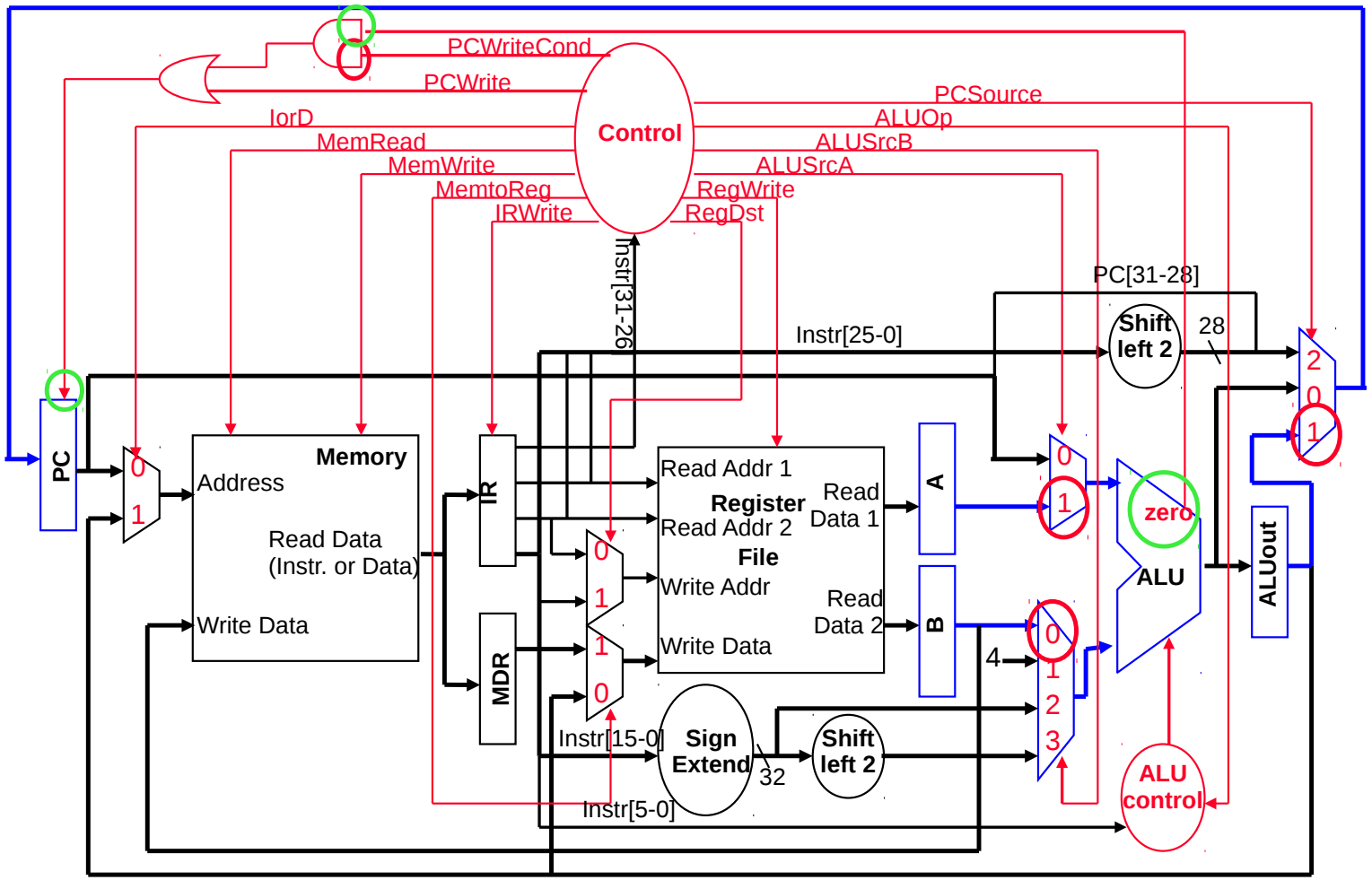
Step 3: Execution, Memory Address Computation or Branch Completion

Branch:

```
if (A == B) PC <= ALUOut
```

- Set ALUSrcA = 1 get operand from A
- Set ALUSrcB = 00 get operand from B
- Set ALUOp = 01 subtraction
- Write ALUOut to PC register

Step 3: Branch

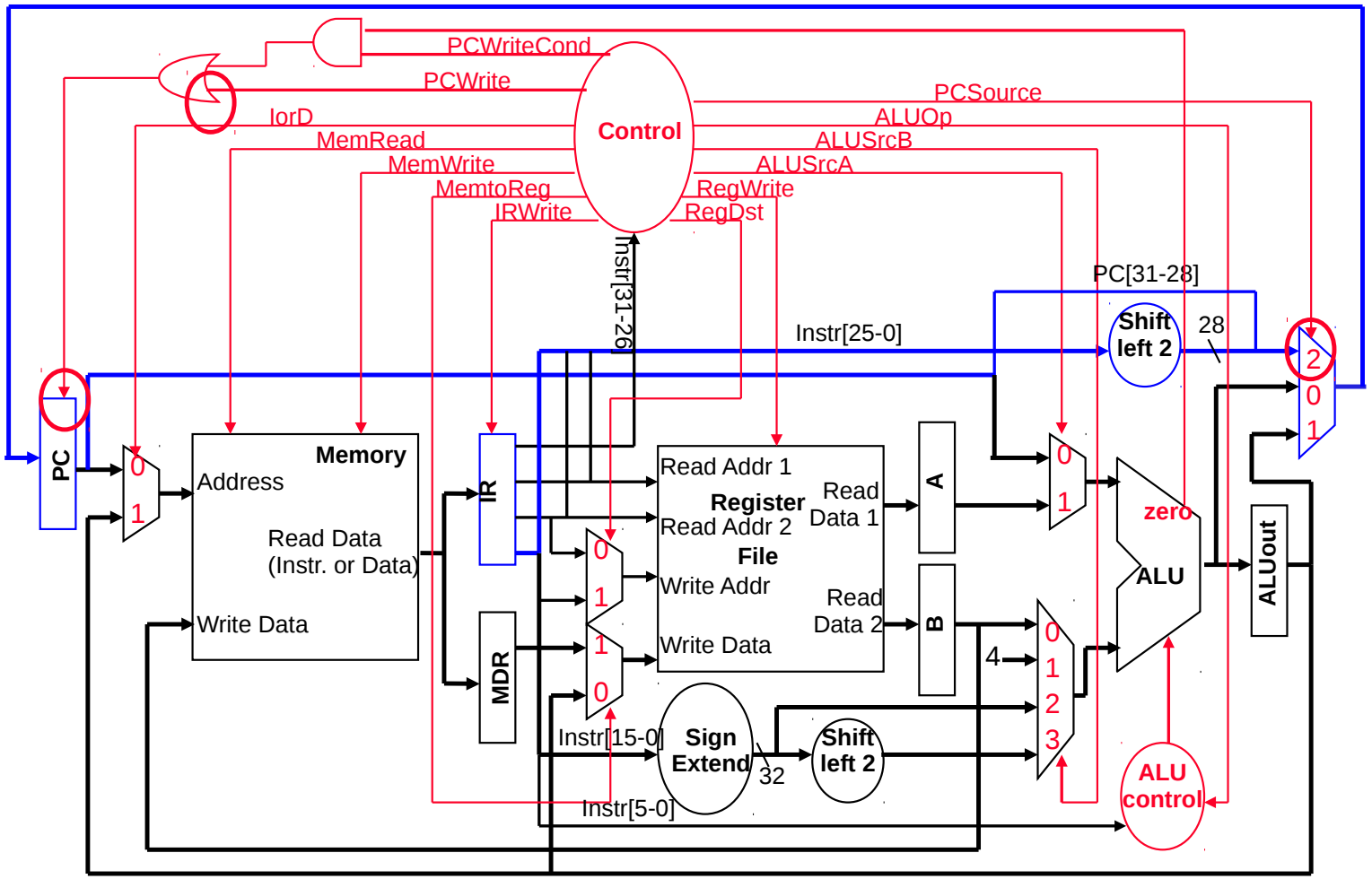


Step 3: Execution, Memory Address Computation or Branch Completion

Jump:

```
PC <= {PC [31:28], (IR[25:0] << 2)}
```

Step 3: Jump

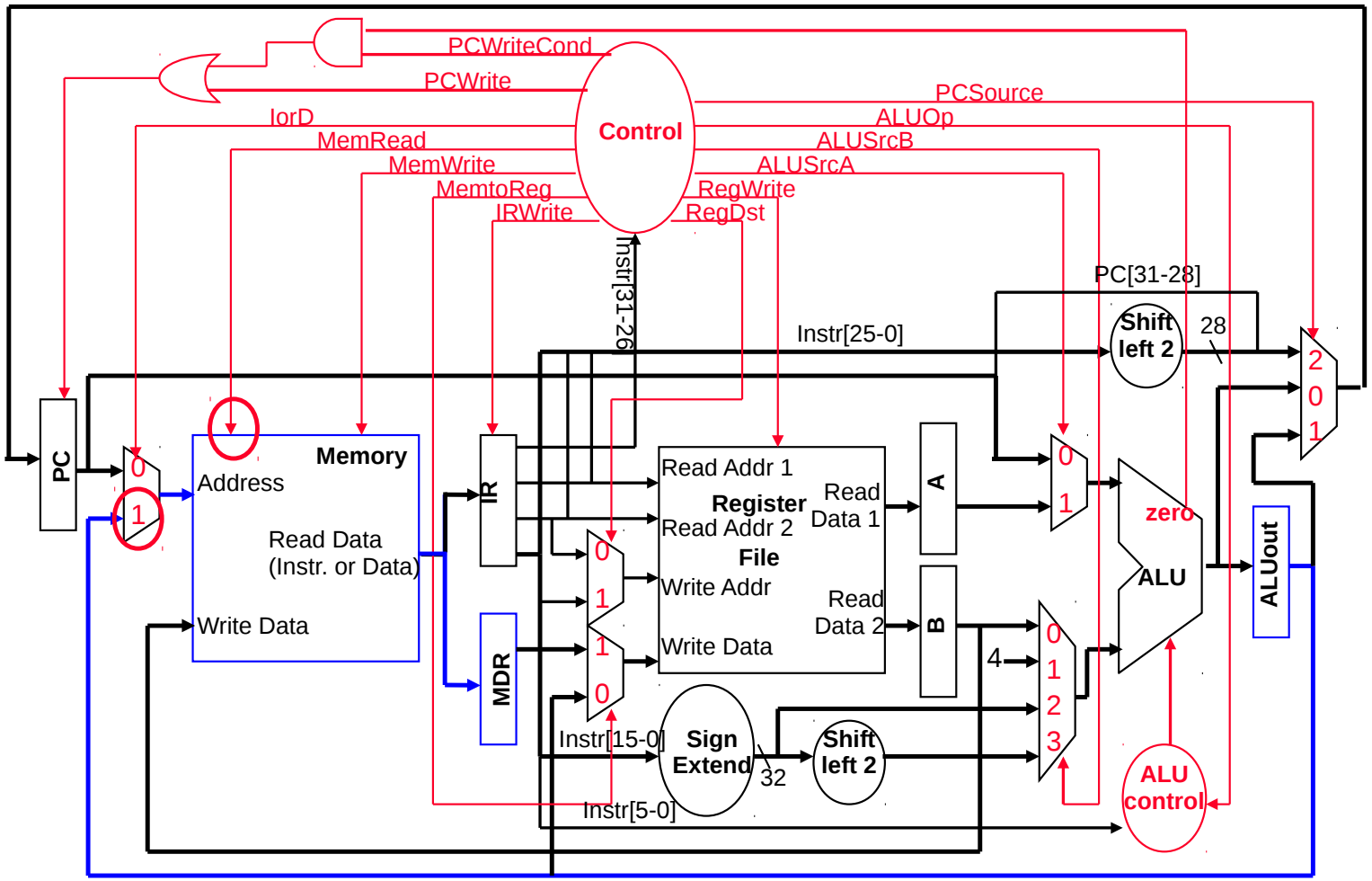

PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
Control
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
Instr[31-26]
PC[31-28]
Instr[25-0]
Shift left 2
28
PC
Memory
Address
Read Data (Instr. or Data)
Write Data
IR
MDR
Read Addr 1
Register File
Read Addr 2
Write Addr
Write Data
Read Data 1
Read Data 2
A
B
4
zero
ALU
ALUout
Instr[15-0]
Sign Extend
32
Shift left 2
Instr[5-0]
ALU control

Step 4: Memory Access or R-type Instruction Completion

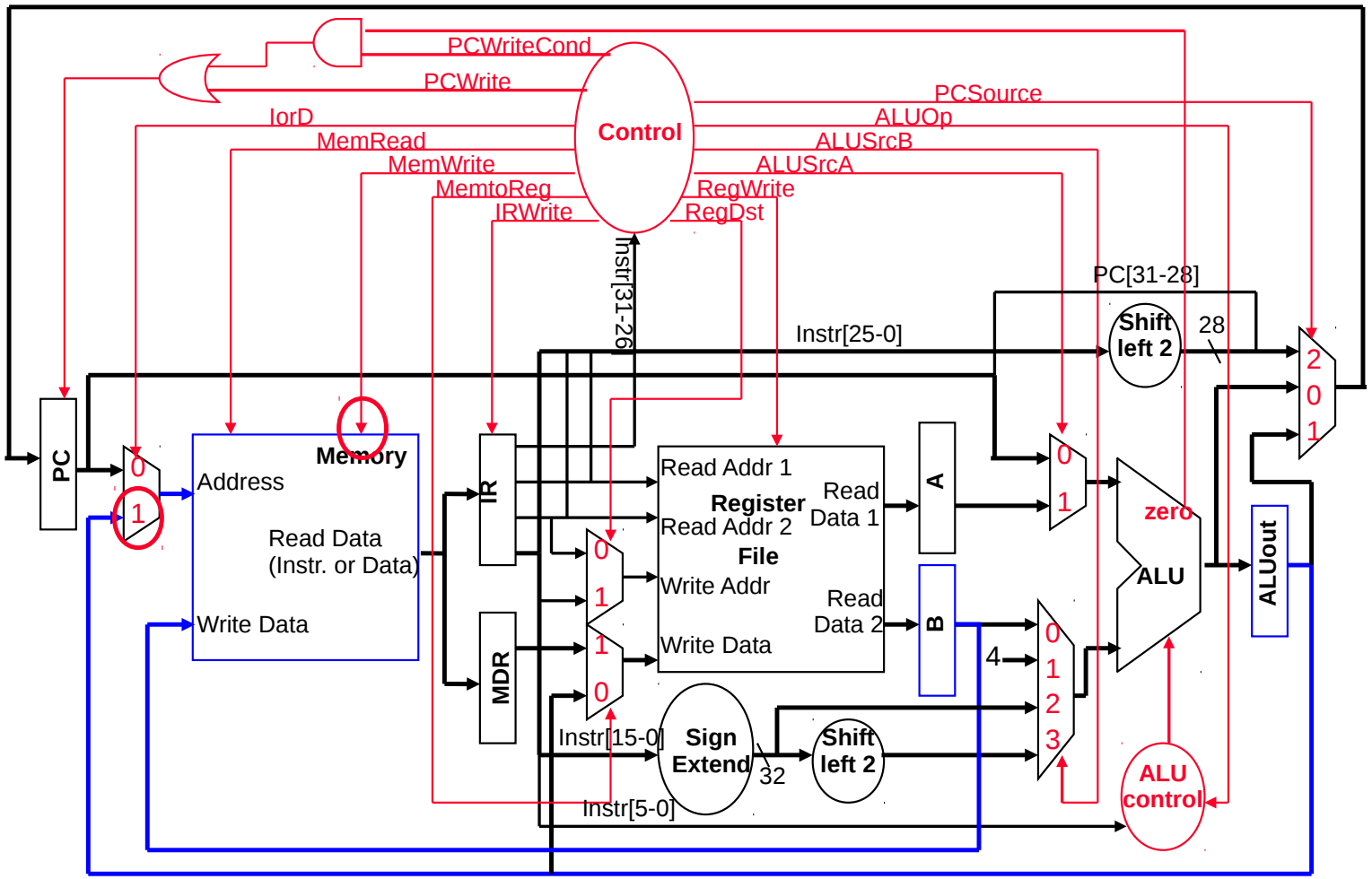
Memory reference:

- ALU controls must remain stable
- Set IorD $= 1$ address from ALU
- load from memory
  `MDR <= memory[ALUOut]`
  - Set MemRead $= 1$
- store to memory
  `memory[ALUOut] <= B`
  - Set MemWrite $= 1$

Step 4: Memory Reference (load word)



Step 4: Memory Reference (save word)



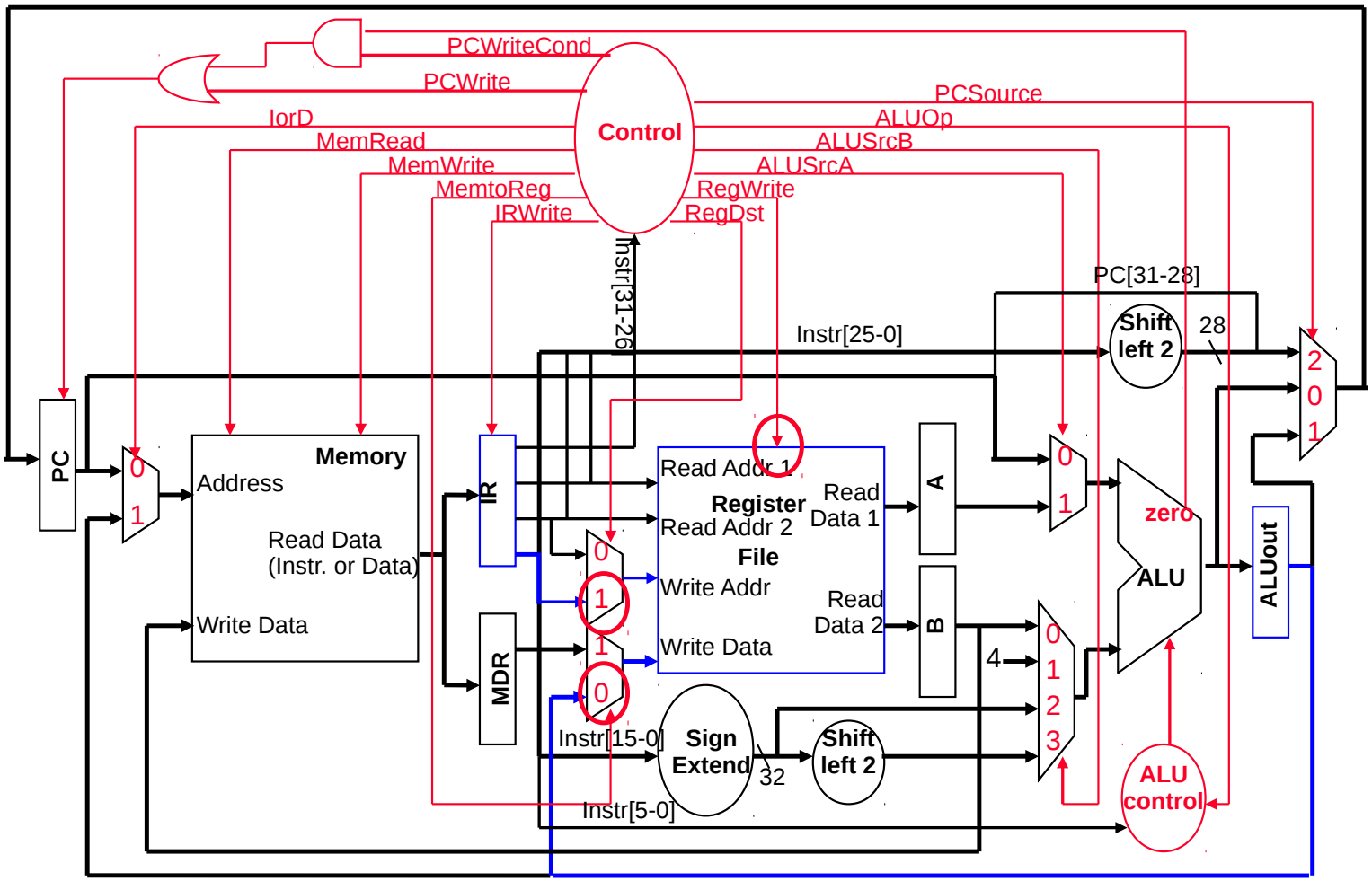
Step 4: Memory Access or R-type Instruction Completion

Arithmetic-logical instruction completion:

```
Register[IR[15:11]] <= ALUOut
```

- Set RegDst = 1 Select write register
- Set RegWrite = 1 Allow write operation
- Set MemToReg = 0 Select ALU data
- ALUOp, ALUSrcA, ALUSrcB = constant

Step 4: Arithmetic-Logical Instruction Completion



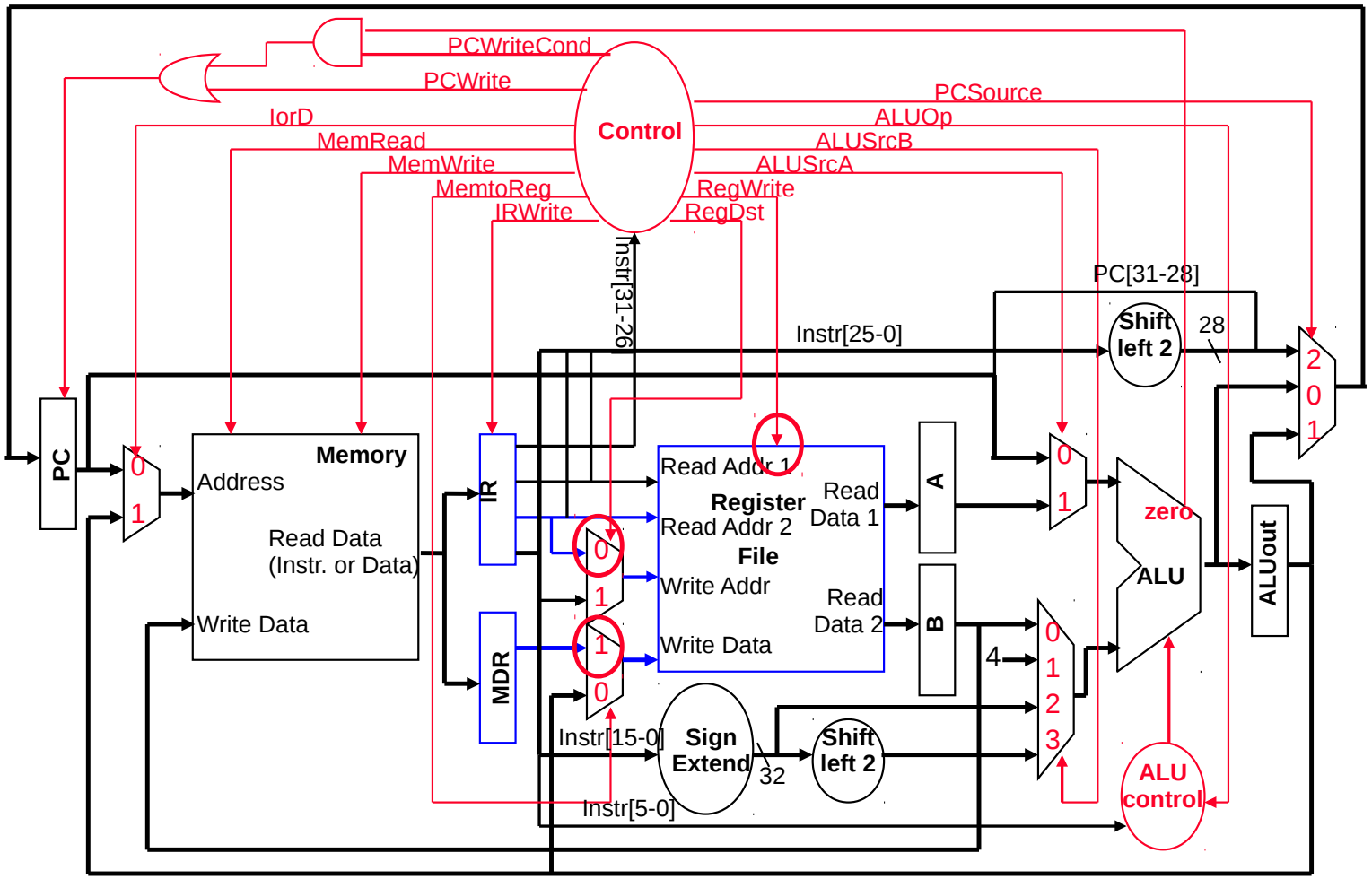
Step 5: Write Back

Write data from memory to the register:

```
Register[IR[20:16]] <= MDR
```

- Set RegDst = 0 Select write rt as target register
- Set RegWrite = 1 Allow write operation
- Set MemToReg = 1 Select Memory data
- ALUOp, ALUSrcA, ALUSrcB = constant

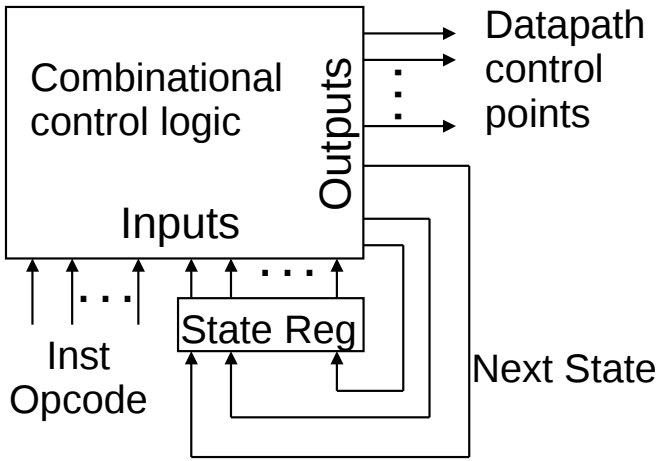
Step 5: Memory Reference (load word)



Multicycle Control Unit

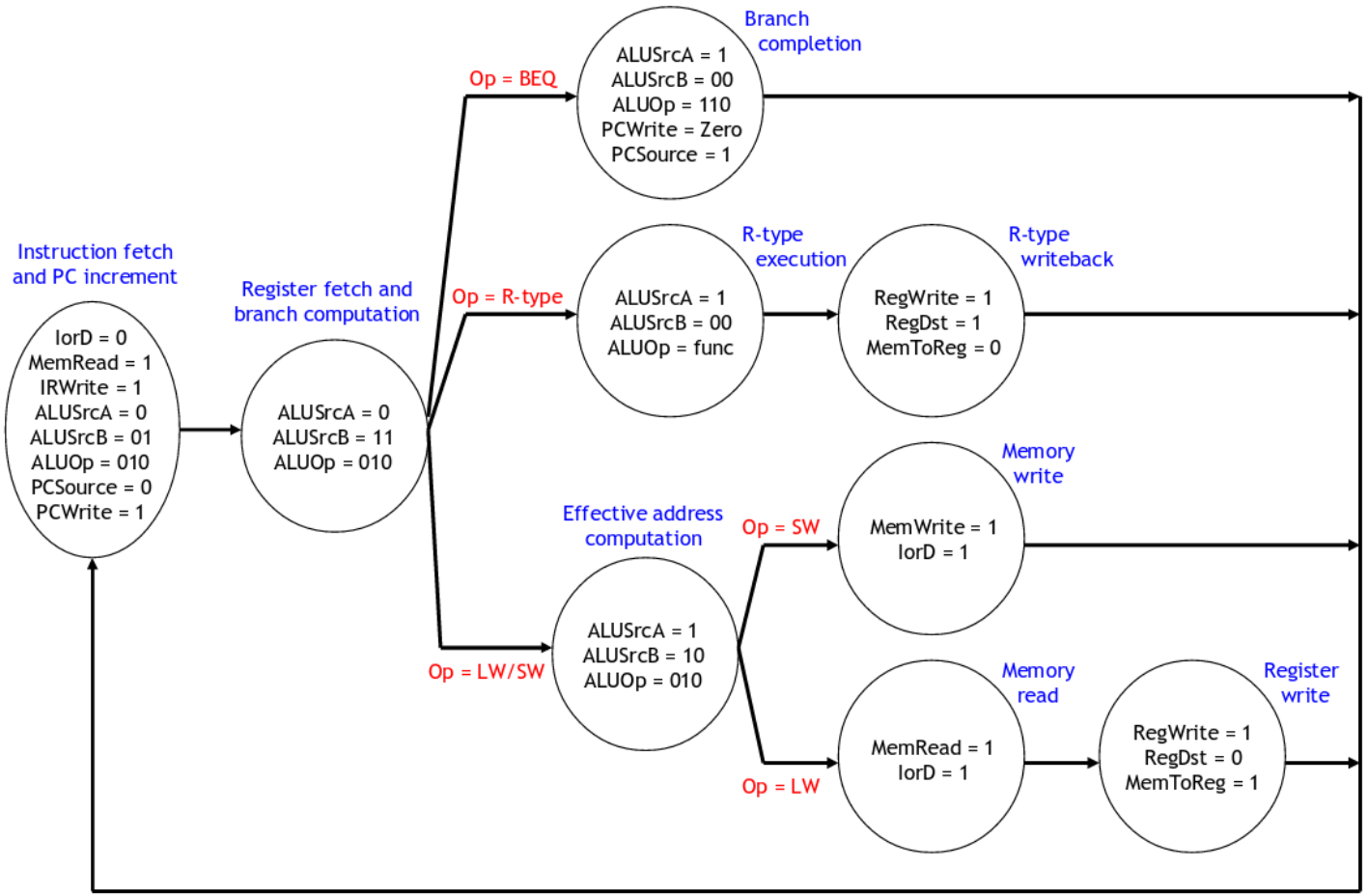
- Not determined solely by the bits in the instruction
  - e.g., op code bits tell what operation the ALU should be doing, but not what instruction cycle is to be done next
- Must use a finite state machine (FSM) for control

- a set of states (current state stored in State Register)
- next state function (determined by current state and the input)
- output function (determined by current state and the input)

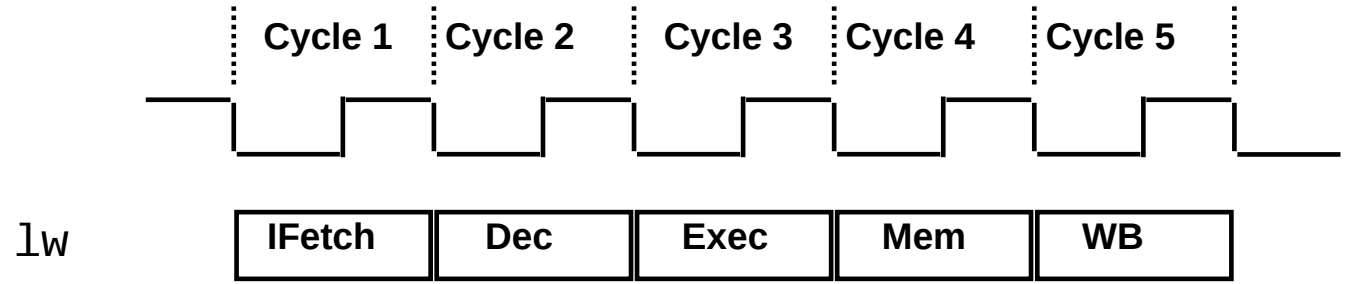


Graphic Representation of FSM

- Common part Instruction specific



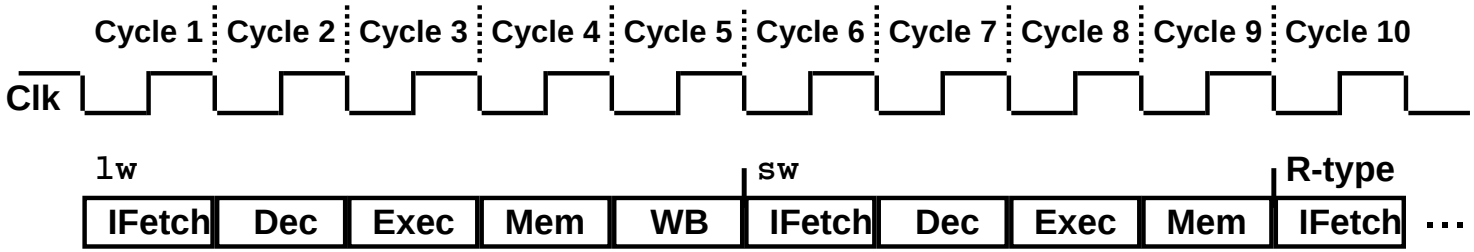
The Five Steps of the Load Instruction



- 1: IFetch: Instruction Fetch and Update PC
- 2: Dec: Instruction Decode, Register Read, Sign Extend Offset
- 3: Exec: Execute R-type; Calculate Memory Address; Branch Comparison; Branch and Jump Completion
- 4: Mem: Memory Read; Memory Write Completion; R-type Completion (RegFile write)
- 5: WB: Memory Read Completion (RegFile write)
- Instructions take $3-5$ cycles

Multicycle Advantages & Disadvantages

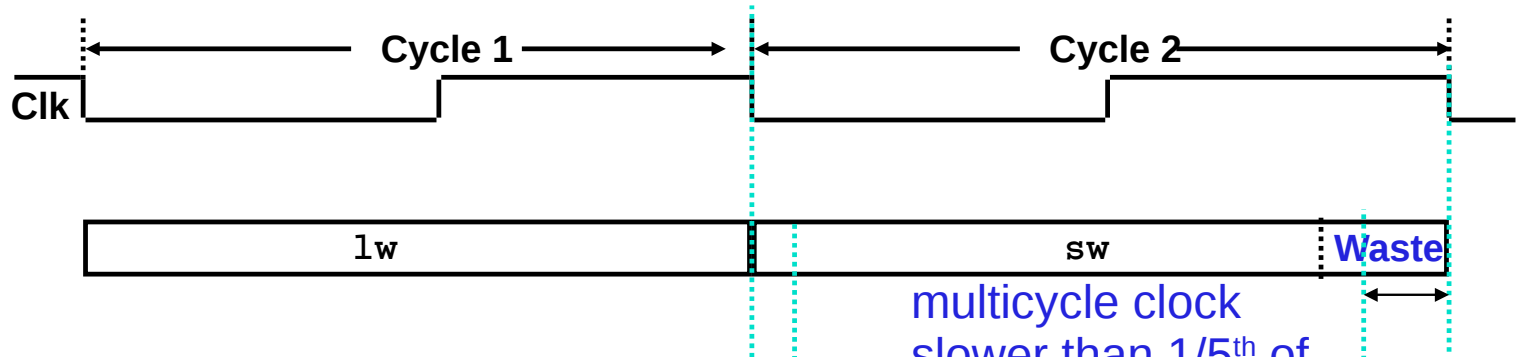
- Uses the clock cycle efficiently – the clock cycle is timed to accommodate the slowest instruction step



- Multicycle implementations allow functional units to be used more than once per instruction as long as they are used on different clock cycles
  But
- Requires additional internal state registers, more muxes, and more complicated (FSM) control

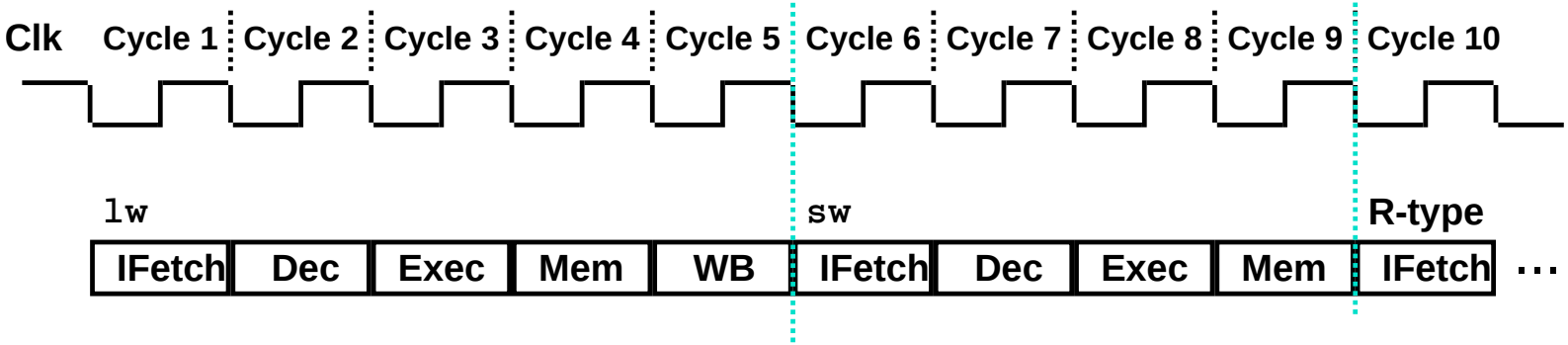
Single Cycle vs. Multiple Cycle Timing

Single Cycle Implementation:



multicycle clock slower than 1/5th of single cycle clock due to state register overhead

Multiple Cycle Implementation:



Summary

- If we understand the instructions
  - We can build a simple processor
- If instructions take different amounts of time, multi-cycle is better
- Datapath implemented using:
  - Combinational logic for arithmetic
  - State holding elements to remember bits
- Control implemented using:
  - Combinational logic for single-cycle implementation
  - Finite state machine for multi-cycle implementation

Pipelining: How Can We Make It Even Faster?

- Split the multiple instruction cycle into smaller and smaller steps
  - There is a point of diminishing returns where as much time is spent loading the state registers as doing the work
- Start fetching and executing the next instruction before the current one has completed
  - Pipelining – (all?) modern processors are pipelined for performance
  - Remember the performance equation:
    CPU time = CPI * CC * IC
- Fetch (and execute) more than one instruction at a time
  - Superscalar processing

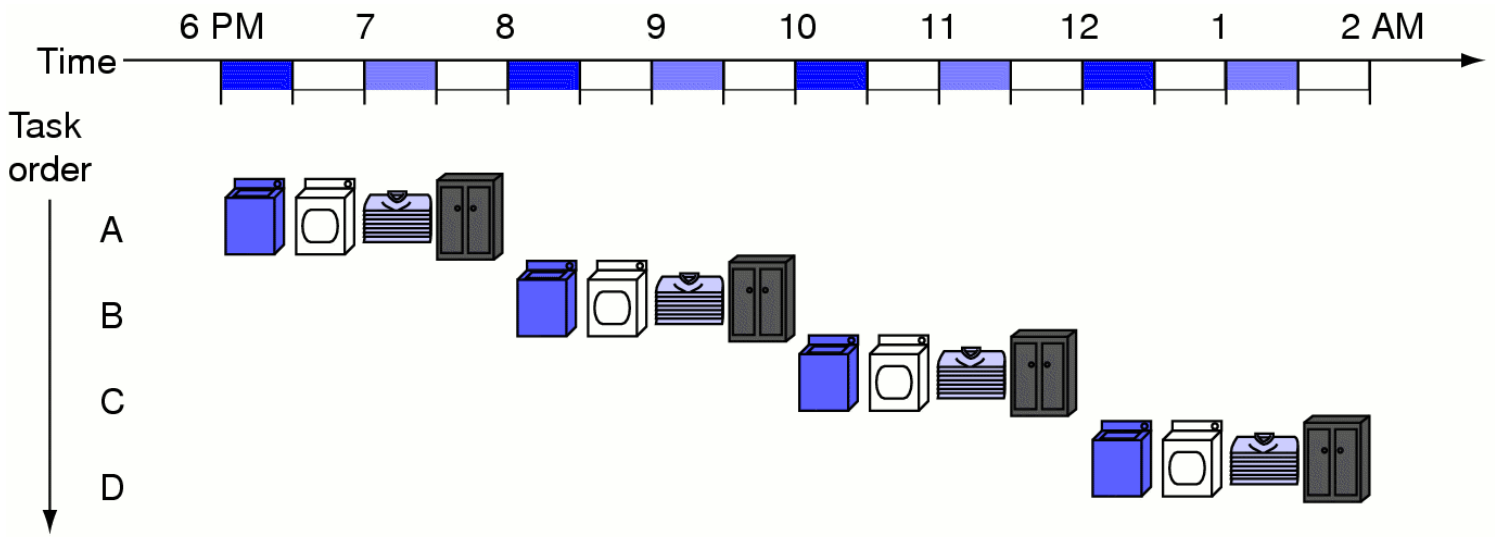
Preconditions

- Instruction set design
  - Instructions (ideally) of equal length
    - enables to fetch in first stage and decode in second
  - Few instruction formats
    - source register at same place for each instruction
    - read register and determine type of instruction
  - Memory operands only in load & store
  - Aligned data: only one memory access/read operation
- Sources of problems
  - Instructions with variable length $\rightarrow$ multiple memory accesses
  - Unaligned data $\rightarrow$ multiple memory access for one data item

An Analogous Example (1)

Laundry problem

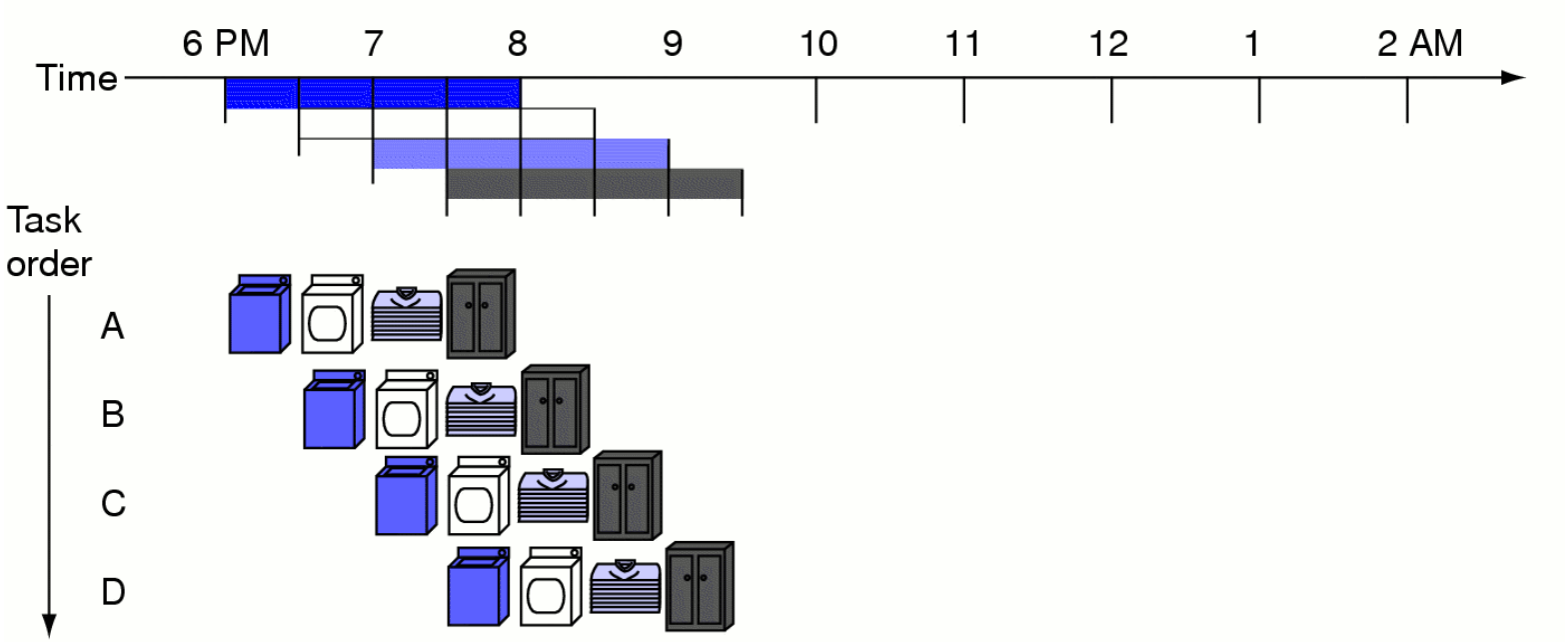
- Four processing stages (wash, dry, fold, put away)
- Identical time (30 minutes)
- Fixed sequence of usage
- Total time for n loads: n * 2 hours



An Analogous Example (2)

Laundry optimization

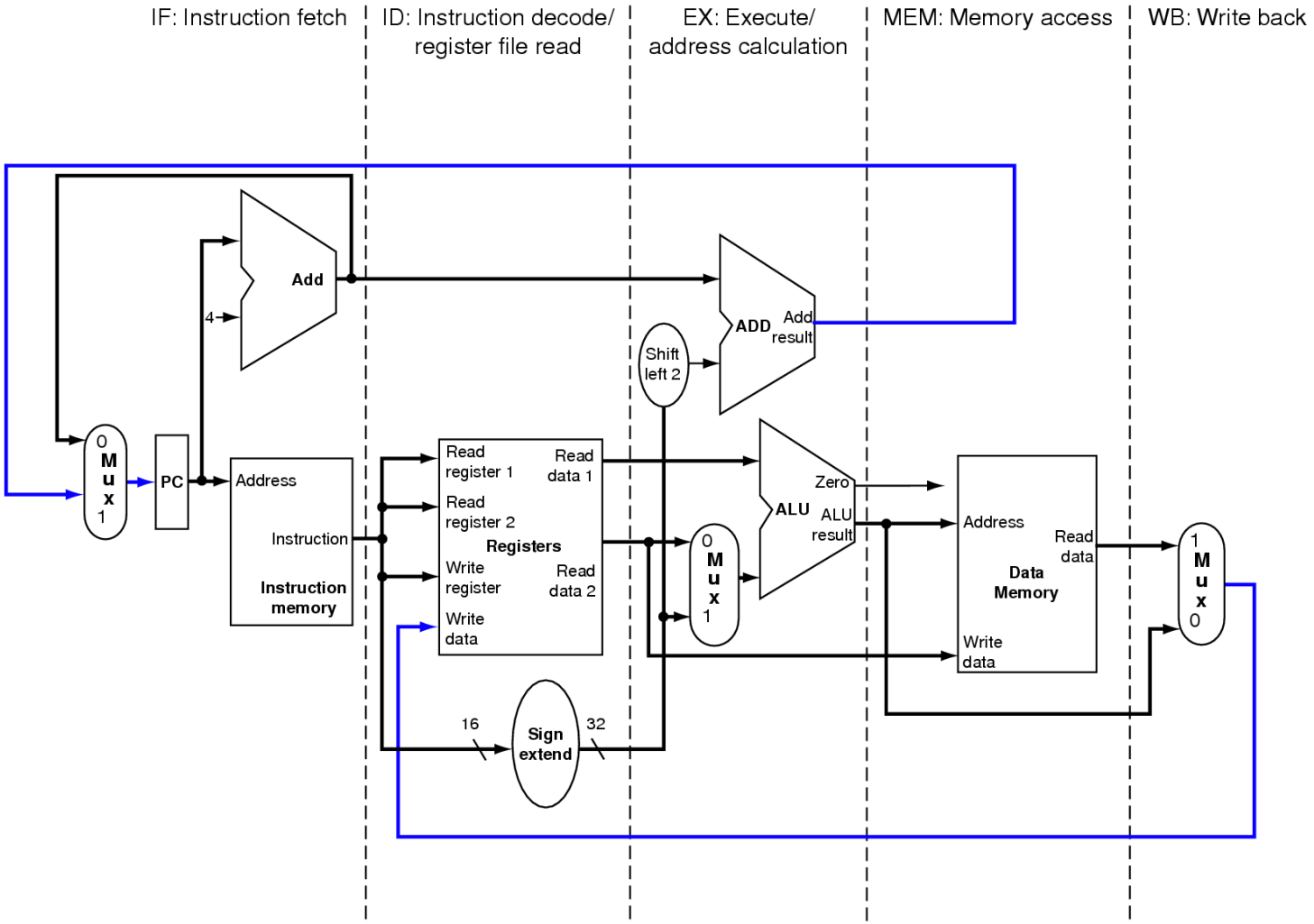
- Units operate independently
- Overlapping use of resources
- Total time, loads: 1 * 2 hours + (n-1) * 1/2 hour = 3.5 hours
- Average time for laundry: 3.5 h / 4 = 52.5 min



An Analogous Example (3)

- All stages operate concurrently
- Many tasks are being done in parallel, pipelining improves throughput of the laundry, while time to complete single load (instructions ...) does not change (latency is not reduced)
- Pipelining is only faster for many loads
  - Far more important metric, because programs execute billions of instructions
- If stages take same amount of time, and if all stages can be used, speedup due to pipelining is equal to number of stages in pipeline
  - But two ifs ..., there is a limit for the length of a pipeline where no further speedup will be seen

Single Cycle Datapath

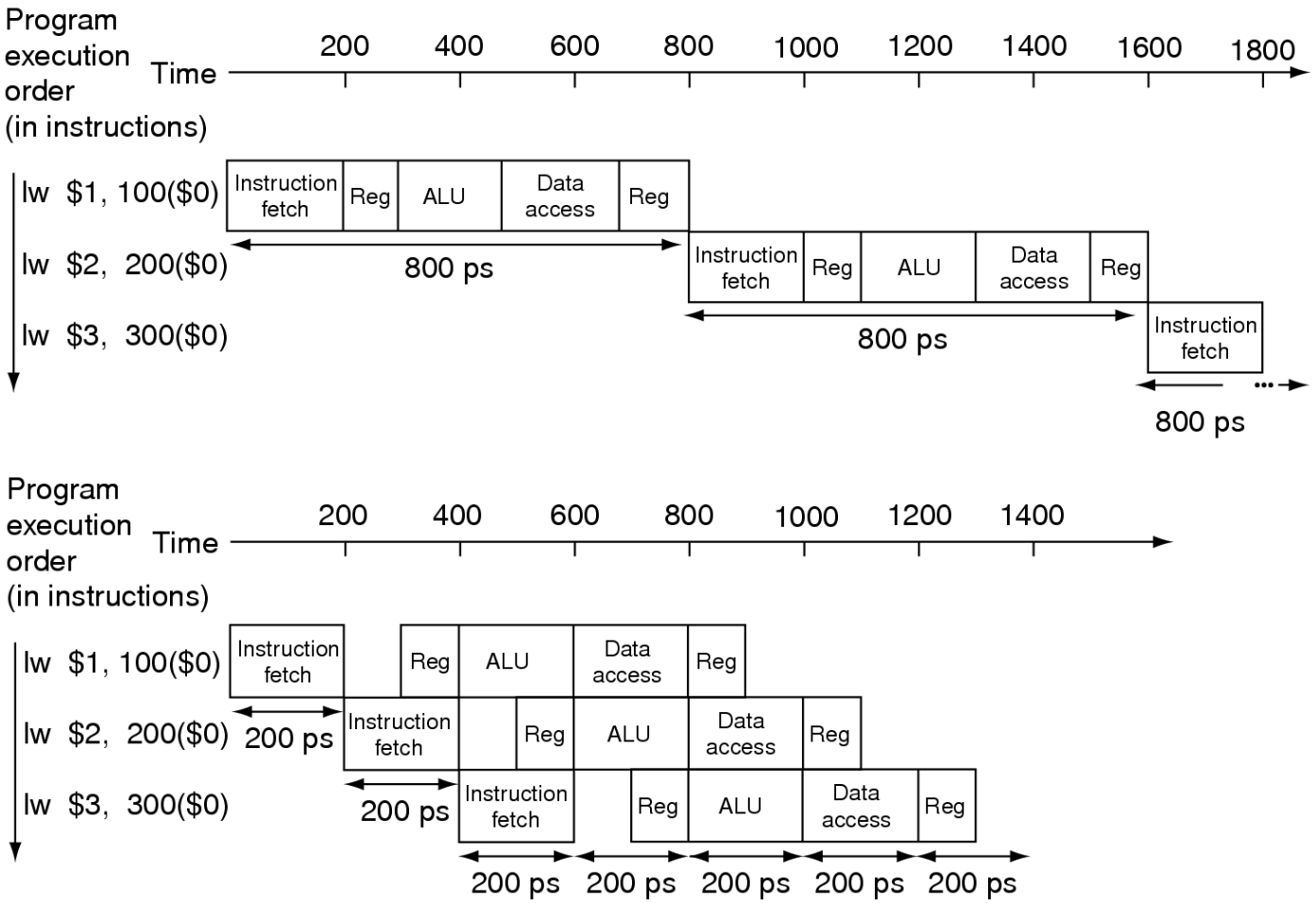


Real Pipeline

- MIPS pipeline steps
  1. IF: Fetch instruction from memory
  2. ID: Read registers while decoding
  3. EX: Execute the operation or calculate an address
  4. MEM: Access an operand in data memory
  5. WB: Write back results in register
- Unequal time for steps (in ps)
  - Single cycle: cycle depends on slowest instruction

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load Word (`lw`)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store Word (`sw`)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (`add, sub, and, or, slt`)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (`beq`)	200 ps	100 ps	200 ps			500 ps

Single Cycle vs. Pipelined Execution



regfile write in first half

regfile read in second half

Pipeline Hazards

- Hazard
  - next instruction cannot execute in the following clock cycle
- Structural hazards
  - competition in accessing hardware resources
  - e.g., accessing the memory at the same time
- Data hazards
  - access to data that is not yet complete
- Control hazards
  - problems in controlling the program flow
  - e.g., branching
  - instruction that was fetched is not the one that was needed

Data Hazards

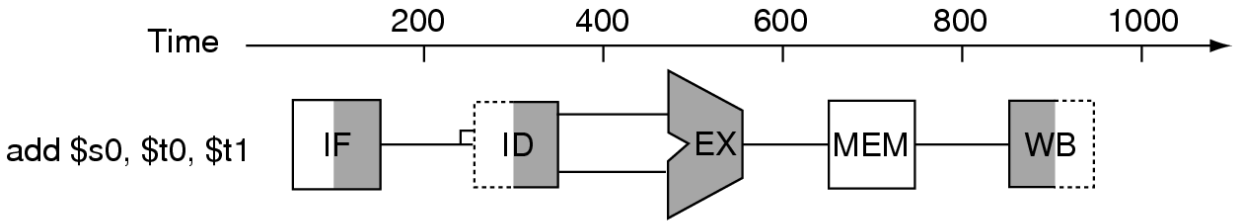
- Access to data that is not yet available/computed

```
1 add $s0, $t0, $t1
2 sub $t2; $s0; $t3
```

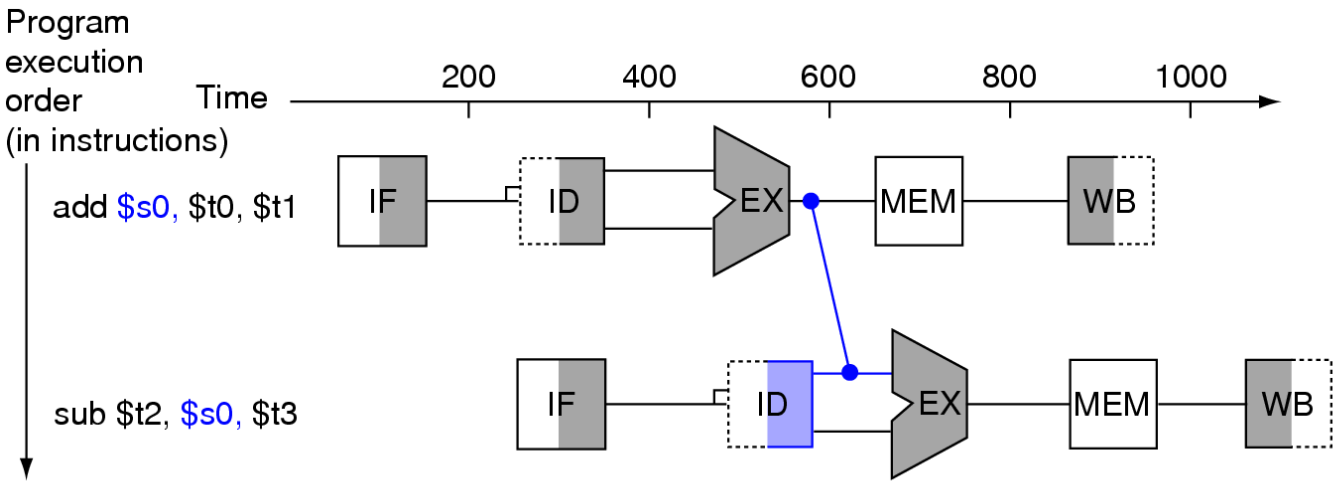
  - Add does not write until 5^{th} stage
  - Sub reads data in stage two
  - Three stalls required
- Solutions
  - Compiler optimization rearranging the instruction sequence
  - Forwarding/Bypassing use results before they are actually written

Forwarding (1)

- Linear execution

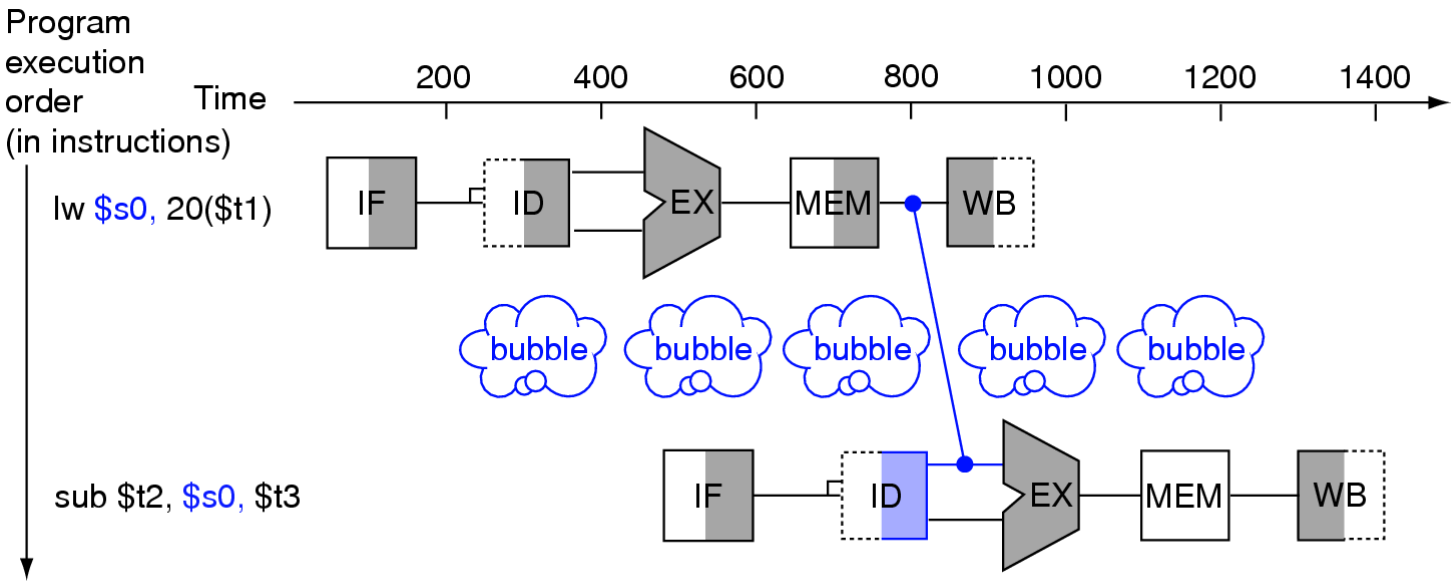


- Direct use of ALU result



Forwarding (2)

- A pipeline stall is a stall in order to resolve a hazard
  - Also called bubble
- Depending on the instructions there are still stalls possible
  - R-format instruction following a load tries to use this data

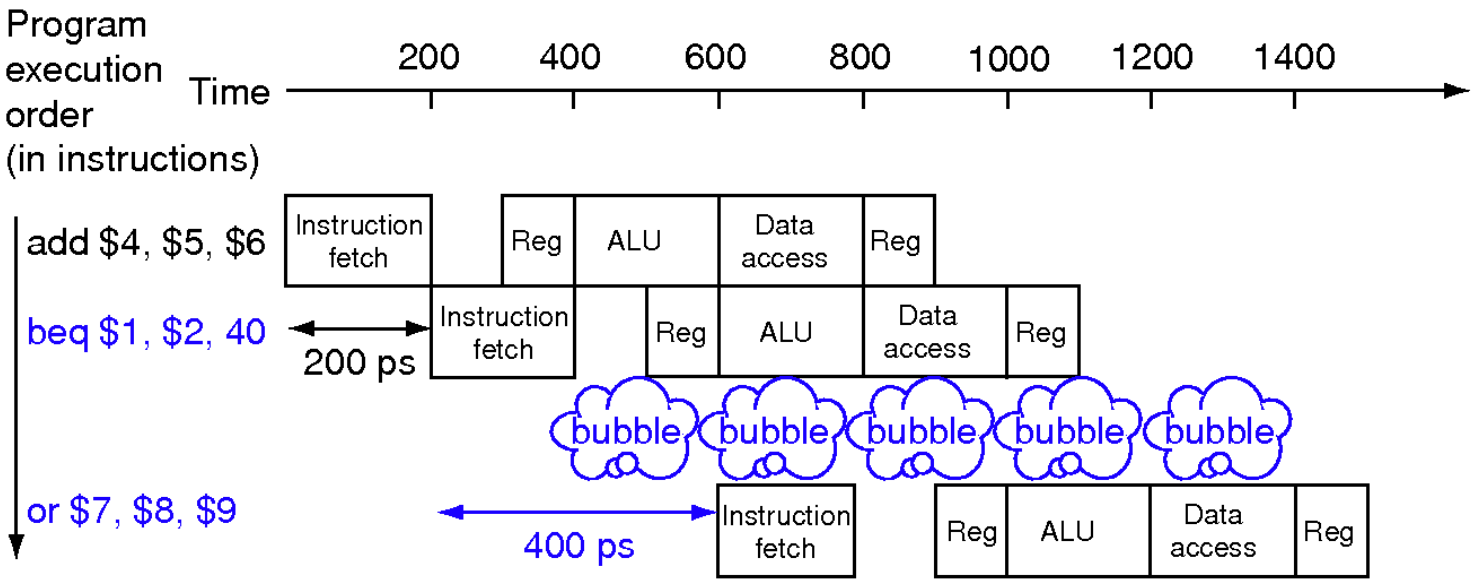


Control Hazards

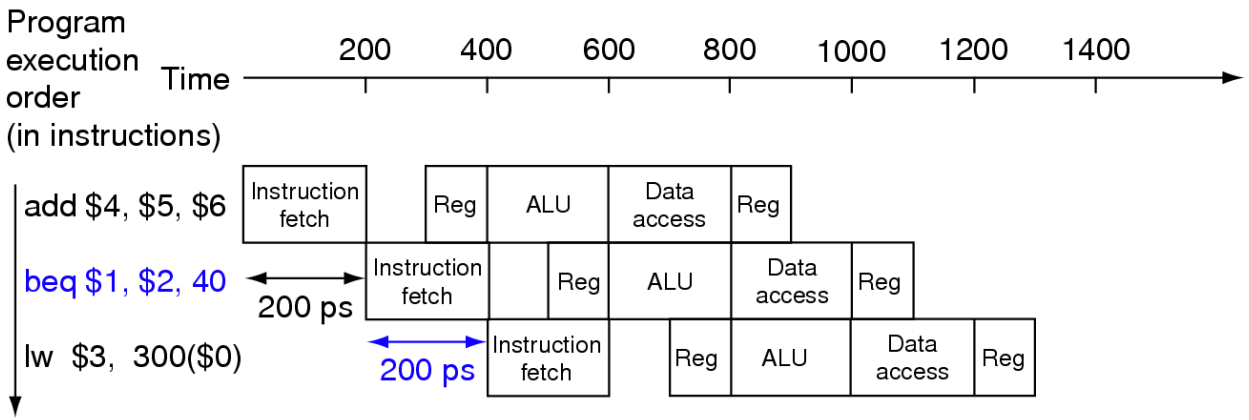
- Proper instruction cannot execute in clock cycle, because fetched instruction is not the one that is needed
  - Normally instruction following branch instruction is being fetched on the very next clock cycle
  - But not known yet which is next instruction
- Possible solutions
  - Insert bubble after each branch
  - Forwarding/Bypassing use results before they are actually written

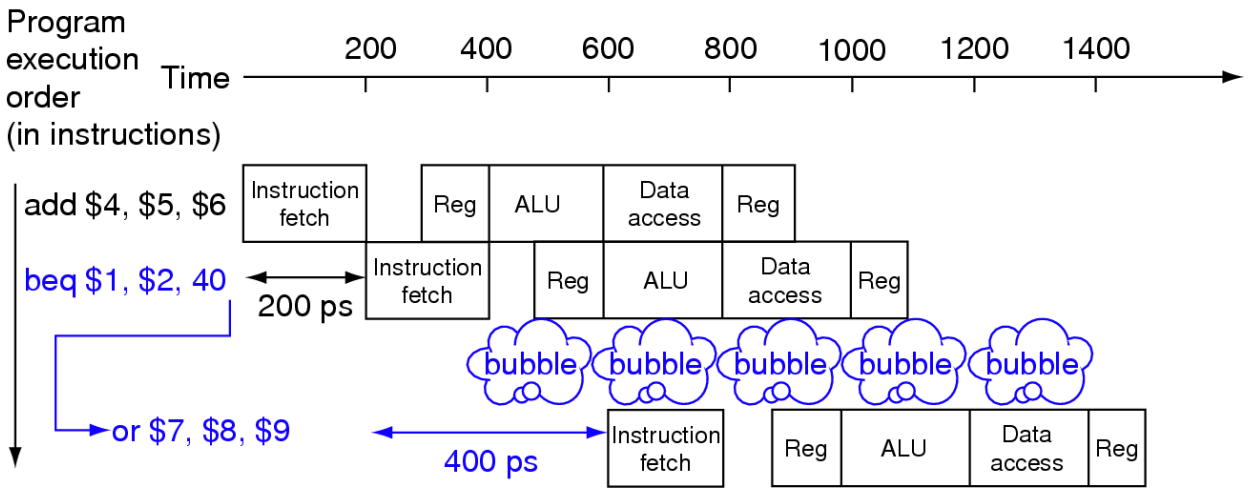
Resolving Control

- Assumption: in stage 2 all branch computations are ready
- Delay by one cycle to wait for result



Branch Prediction





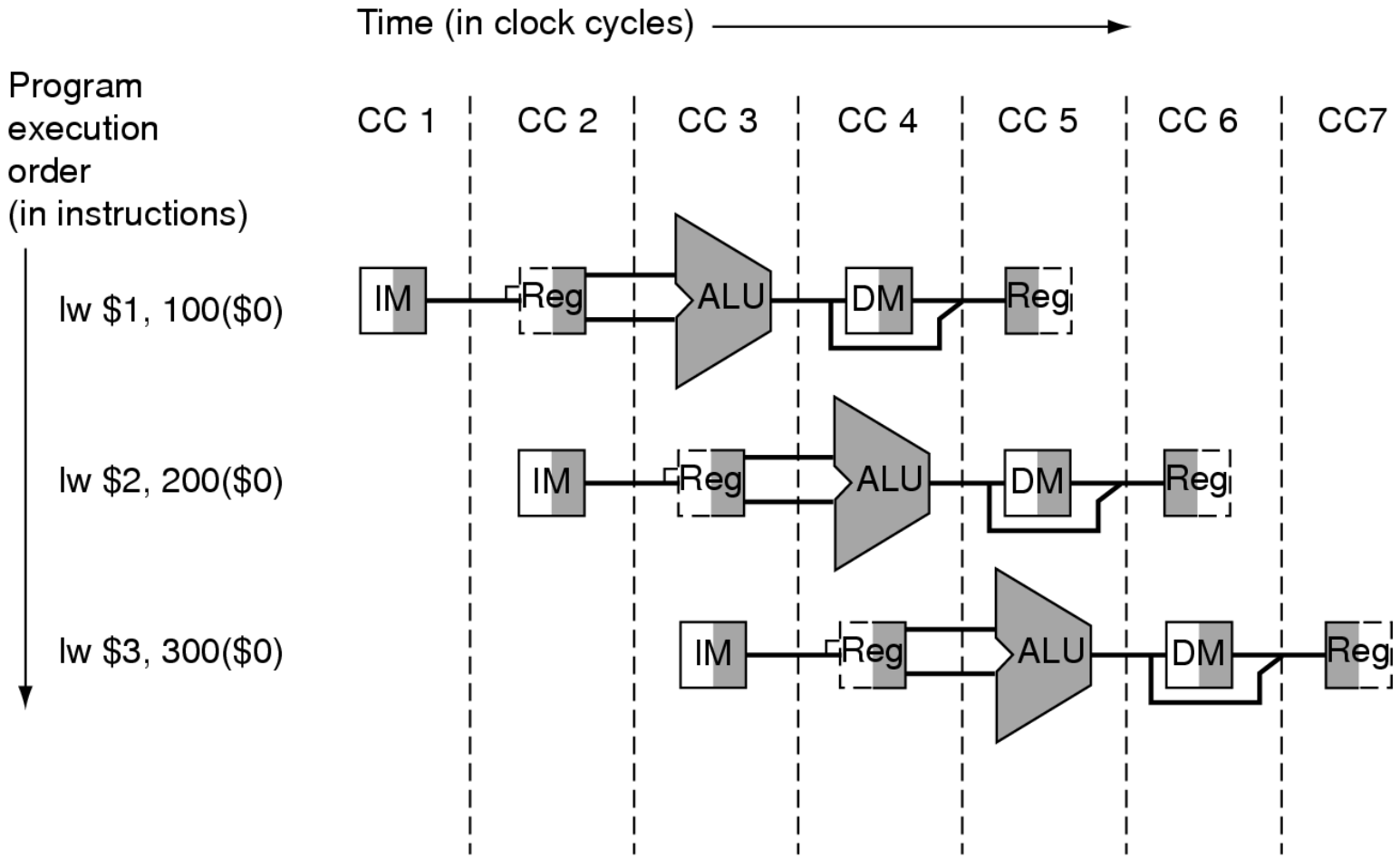
Pipelined Datapath

- Reuse of functional units
- Additional hardware for:
  - Separation of pipeline steps
  - Functional units if used by different instructions at the same time
- Extended control for:
  - Strict sequentialization of instruction
  - Check for hazards
  - Remove hazards – introduce stalls

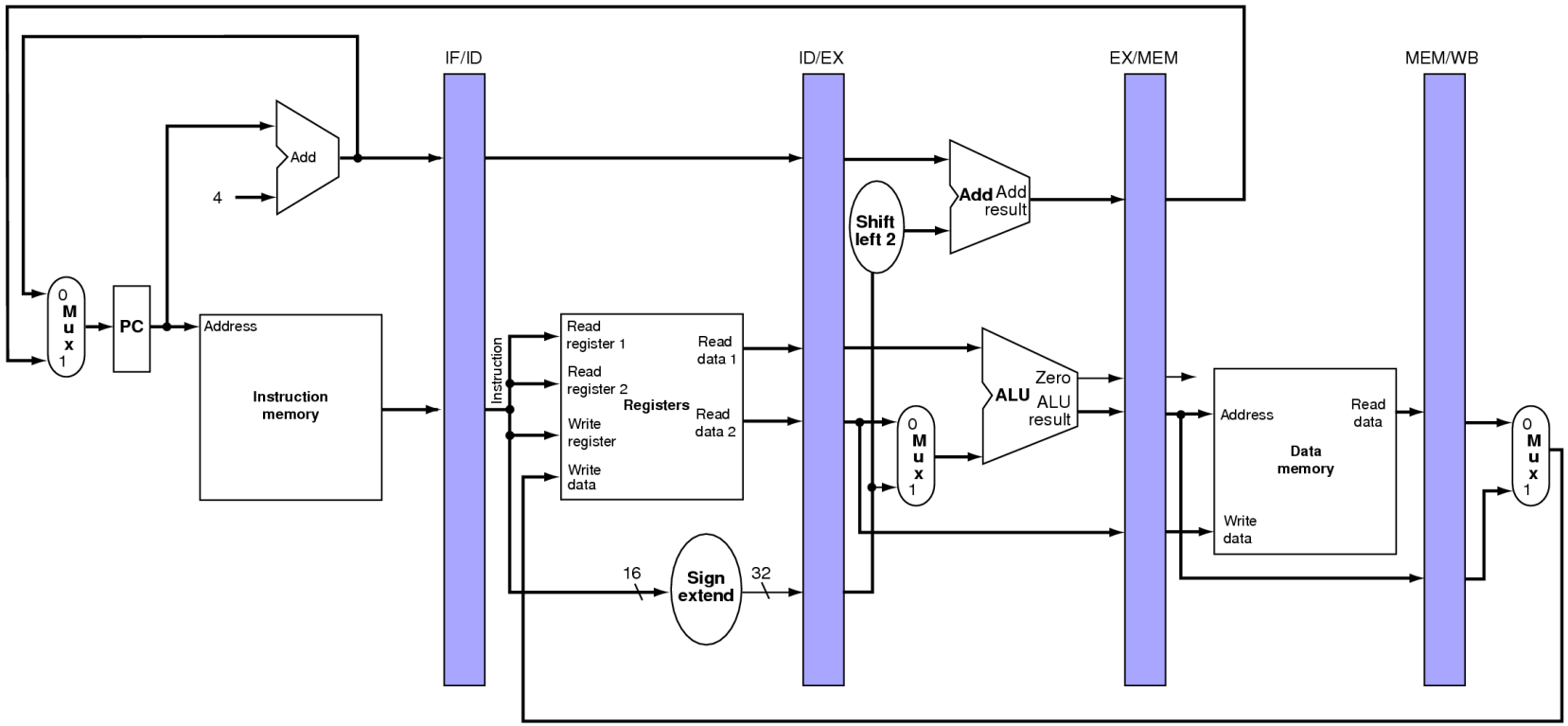
Problems

- Normally left-to-right flow, but exceptions
  - Write back data to the register file
    - Creates a data hazard
  - Selection of the next PC
    - Creates a control hazard
- Solution:
  - Have a separate data path for each instruction $\rightarrow$ high hardware effort $\rightarrow$ not affordable
  - Chop the data path into small chunks
  - Keep everything what belongs together in one chunk
  - Introduce registers for separating the stages

Pipelined Version of Datapath (1)



Pipelined Version of Datapath (2)



- Colored pipeline registers separate each pipeline stage
- Register width varies

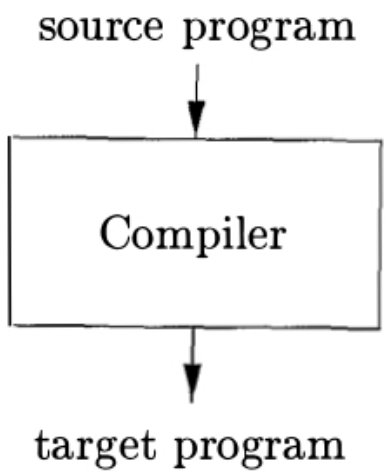
Summary

- Some instructions do not require the complete data path
- No information transfer from one pipeline stage to another is possible except through the pipeline registers
- Everything that happened in any previous stage will be overwritten
- Each logical component (instruction memory, register read ports, ALU, data memory, register write port) can be used only within a single pipeline stage
- Correction for load word instruction required:
  - where is the information on the write register

Part IV: Programming Languages and Compilation

Language Processors (1)

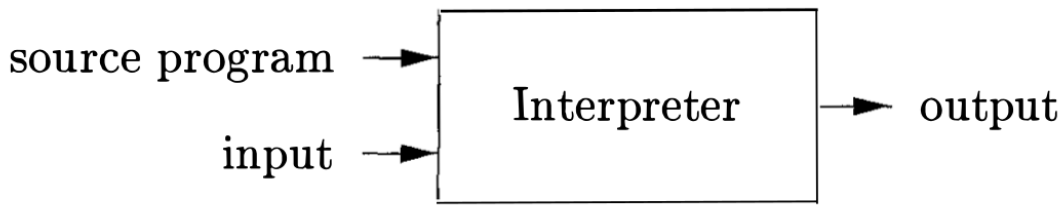
- Before a program can be run, it first must be translated into a form in which it can be executed by a computer
- The software systems that do this translation are called compilers
- Simply, a compiler is a program that can read a program in one language - the source language - and translate it into an equivalent program in another language - the target language
- The compiler should report any errors in the source program that are detected during the translation process



Language Processors (2)

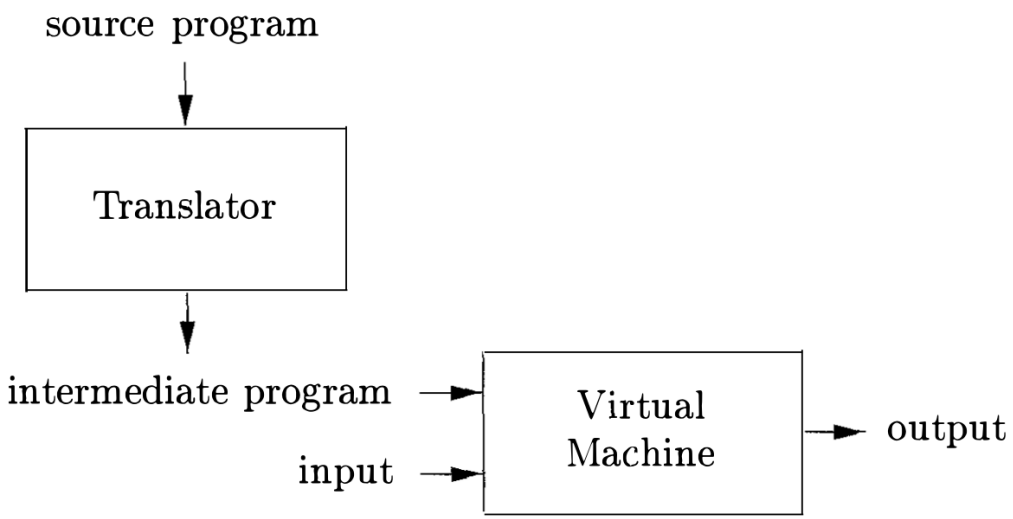
If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs

An interpreter produces a target program as a translation, it appears to directly execute the operations specified in the source program on inputs supplied by the user



Language Processors (3)

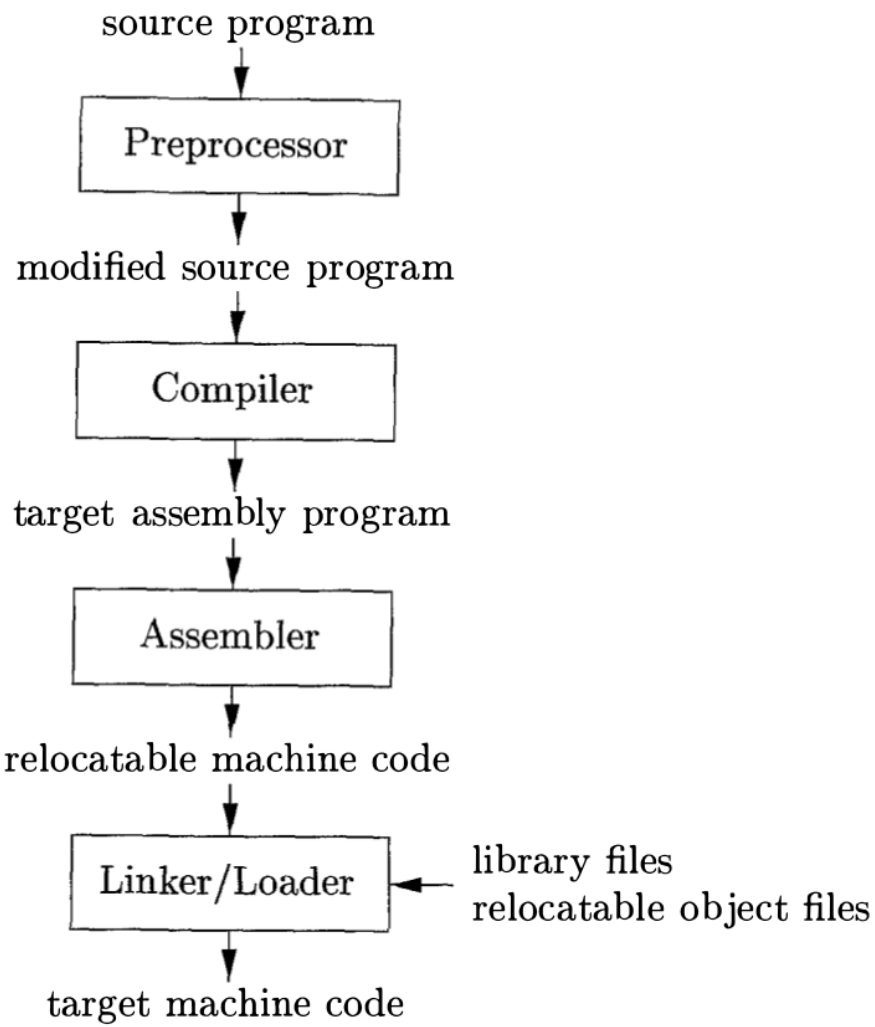
The machine-language target program produced by a compiler is much faster than an interpreter at mapping inputs to outputs. An interpreter can give better error diagnostics than a compiler.
Java language processors combine compilation and interpretation and are therefore called hybrid compilers



Other Language Processors

- Preprocessor: collects the source program is sometimes entrusted to a separate program, also expands shorthands, called macros, into source language statements
- Assembler: the compiler may produce an assembly-language program as its output, because assembly language is easier to produce as output and is easier to debug, the assembler then produces relocatable machine code as its output
- Linker: resolves external memory addresses, where the code in one file may refer to a location in another file
- Loader: puts together all of the executable object files into memory for execution

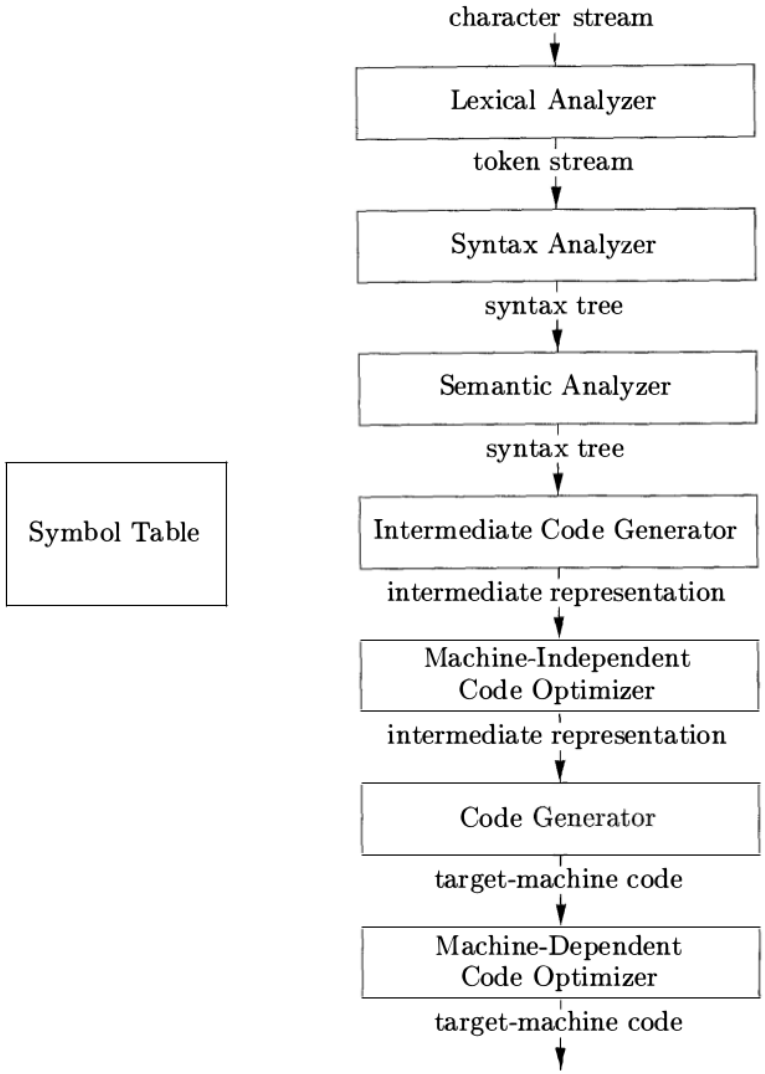
A Language-Processing System



Analysis and Synthesis

- Analysis: breaks up the source program into constituent pieces and imposes a grammatical structure on them
  - This structure is used to create an intermediate representation of the source program
  - If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action
  - Information is collected about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis
- Synthesis: constructs the desired target program from the intermediate representation and the information in the symbol table
  - The analysis part is often called the front end of the compiler
  - The synthesis part is the back end

Phases of Compilation



Lexical Analysis

- Is called lexical analysis or scanning
- The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes
- For each lexeme, the lexical analyzer produces as output a token of the form `<token-name, attribute-value>` that it passes on to the subsequent phase, syntax analysis
- In the token, the first component `token-name` is an abstract symbol that is used during syntax analysis, and the second component `attribute-value` points to an entry in the symbol table for this token

Translation Example (1)

Assume source program contains

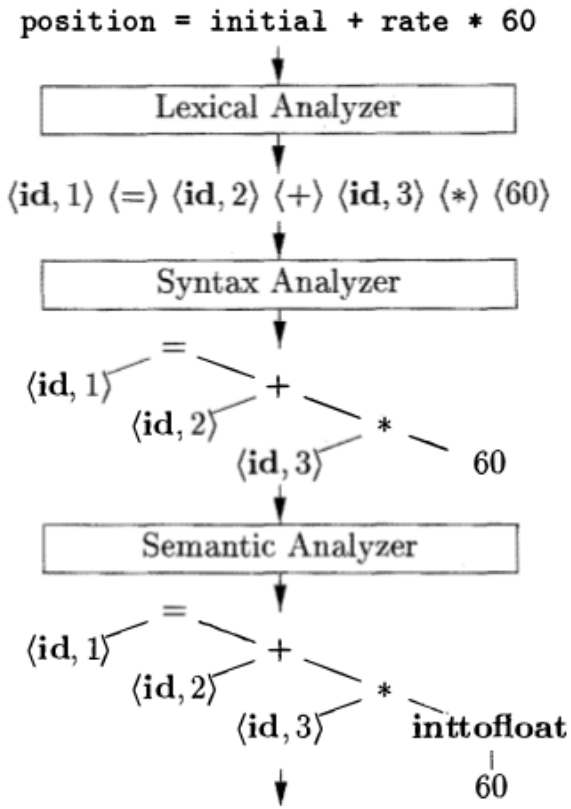
`position = initial + rate * 60`

- `position` is a lexeme that would be mapped into a token `<id, 1>`, where `id` is an abstract symbol standing for identifier and `1` points to the symbol table entry for `position`
- `=` is a lexeme that is mapped into the token `<=>`
- `initial` is a lexeme that is mapped into the token `<id, 2>`, where `2` points to the symbol-table entry for `initial`
- `+` is a lexeme that is mapped into the token `<+>`
- `rate` is a lexeme that is mapped into the token `<id, 3>`, where `3` points to the symbol-table entry for `rate`
- `*` is a lexeme that is mapped into the token `<*>`
- `60` is a lexeme that is mapped into the token `<60>`

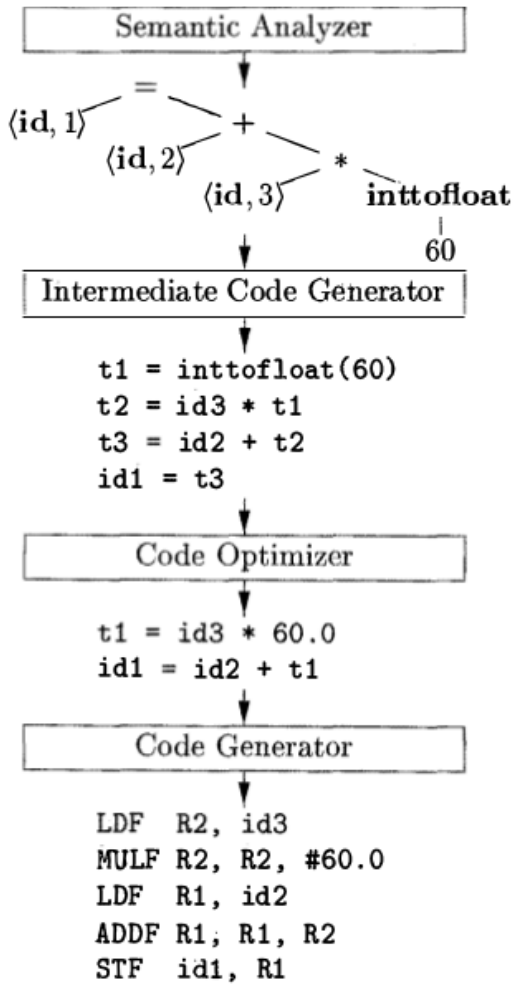
Translation Example (2)

1	position	···
2	initial	···
3	rate	···

SYMBOL TABLE



Translation Example (3)



Syntax Analysis

- The second phase of the compiler is syntax analysis or parsing
- It uses the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure
- It is called the syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation
- The syntax tree shows the order in which the operations in the assignment `position = initial + rate * 60` are to be performed

Semantic Analysis

- The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition
- It gathers type information and saves it in the syntax tree or the symbol table, for the intermediate-code generation
- One part is type checking, where the compiler checks that each operator has matching operands (e.g., an array index has to be an integer)

Intermediate Code Generation

- During translation, a compiler may construct one or more intermediate representations, which can have a variety of forms
- After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation
- This representation should have two important properties:
  - it should be easy to produce
  - it should be easy to translate into the target machine

Example: Intermediate Code Generation

- Intermediate form called three-address code, which consists of a sequence of assembly-like instructions with three operands per instruction
- Each operand can act like a register
- The output of the intermediate code generator from previous example consists of the three-address code sequence

```
1 t1 = inttofloat(60)
2 t2 = id3 * t1
3 t3 = id2 + t2
4 id1 = t3
```

- Each three-address assignment instruction has at most one operator on the right side
- The compiler must generate a temporary name to hold the value computed by a three-address instruction
- Some "three-address instructions" like the first and last in the sequence above, have fewer than three operands

Code Optimization

- Attempts to improve the intermediate code so that better target code will result
- Better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power
- The optimizer can deduce that the conversion of `60` from integer to floating point can be done once and for all at compile time, so the `inttofloat` operation can be eliminated by replacing the integer `60` by the floating-point number `60.0`
- Moreover, `t3` is used only once to transmit its value to `id1` so the optimizer can transform the previous code into the shorter sequence

```
1 t1 = id3 * 60.0
2 id1 = id2 + t1
```

Code Generation (1)

- The code generator takes as input an intermediate representation of the source program and maps it into the target language
- If the target language is machine code, registers or memory locations are selected for each of the variables used by the program
- Intermediate instructions are translated into sequences of machine instructions that perform the same task
- A crucial aspect of code generation is the judicious assignment of registers to hold variables

Code Generation (2)

- Using `R1` and `R2`, the intermediate code might get translated into the machine code

```
1 LDF R2, id3
2 MULF R2, R2, #60.0
3 LDF R1, id2
4 ADDF R1, R1, R2
5 STF id1, R1
```

- The first operand of each instruction specifies a destination
- The `F` in each instruction tells us that it deals with floating-point numbers