ECSE324 : Computer Organization

Processor Implementation
Chapters 5 & 6

Christophe Dubach

Fall 2021

Revision history: Warren Gross – 2017, Christophe Dubach – W2020&F2020, Brett H. Meyer – W2021, Christophe Dubach – F2021

Some material from Hamacher, Vranesic, Zaky, and Manjikian, *Computer Organization and Embedded Systems, 6 th ed*, 2012, McGraw Hill, and “Introduction to the ARM Processor using Altera Toolchain.”

Timestamp: 2021/11/19 10:23:00

Disclaimer

Lectures are recorded live and posted unedited on *MyCourses* on the same day.

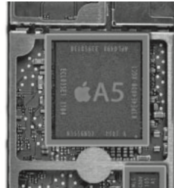
It is possible (and even likely) that I will (sometimes) make mistakes and give incorrect information during the live lectures. If you have any doubts, please check the textbook, or ask on the online forum for clarification.

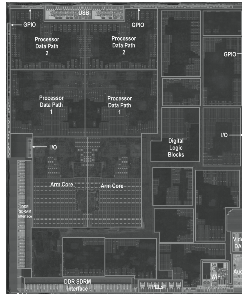
Introduction

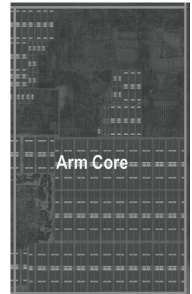
The Processor

The processor reads program instructions from the computer's memory and executes them.

- Each instruction is first fetched from memory
- Then, the instruction is decoded (interpreted); its operands are read
- Finally, the instruction is executed, and any results are stored

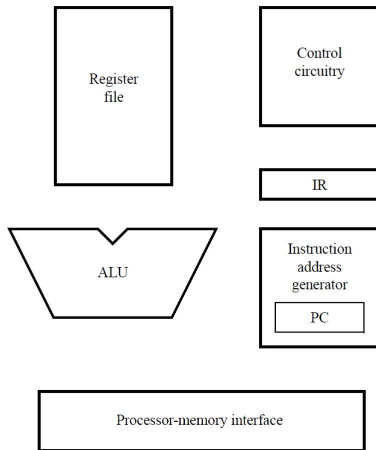






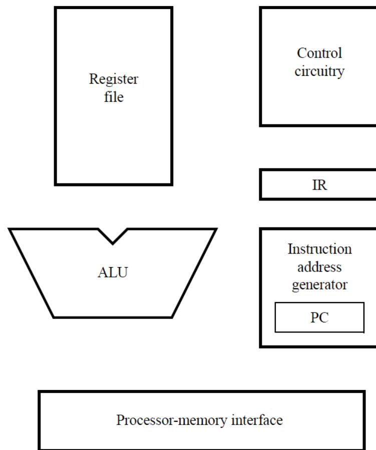
Processor Building Blocks

- PC holds the address of the next instruction to be fetched, decoded, executed
- This instruction is saved in the instruction register IR: `IR` $\leftarrow$ `Mem[PC]`
- The instruction address generator updates PC: `PC` $\leftarrow$ `PC + 4`*
- *Control circuitry* decodes the instruction and generates the signals that direct the *datapath* (everything else)



Processor Building Blocks

- PC holds the address of the next instruction to be fetched, decoded, executed
- This instruction is saved in the instruction register IR: IR $\leftarrow$ Mem[PC]
- The instruction address generator updates PC: PC $\leftarrow$ PC + 4*
- *Control circuitry* decodes the instruction and generates the signals that direct the *datapath* (everything else)



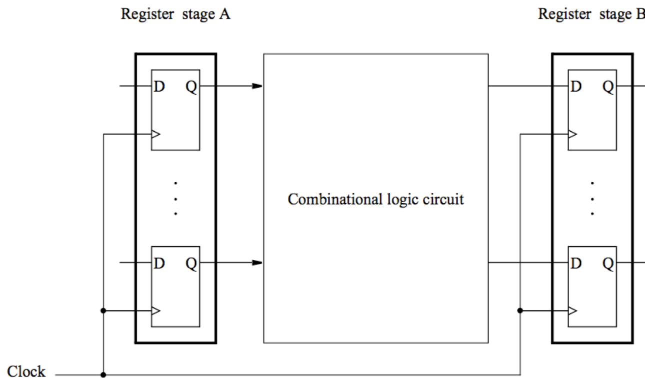
* a mux selects between PC + 4 and PC + disp (generated by a branch instruction and associated hardware)

Datapath Design

Textbook§5.1-5.4

Data Processing Hardware

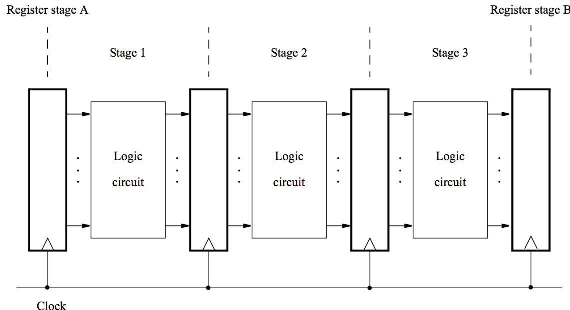
The contents of register A are processed, and the result is saved in register B.



The clock period is determined by the delay through the combinational logic.

Data Processing Hardware

Combinational logic can be divided into simpler sub-circuits that are cascaded into multiple stages.



- n stages: n clock cycles to complete the operation
- Clock period can be shorter: $1/n$
- Pipelining increases throughput $n\times$! (more on this later)
- Effective latency becomes 1 clock cycle!

Instruction Execution

In RISC machines, all* instructions are executed in the same number of steps.

- Each step is carried out in a separate hardware stage
- We will look at a five-stage design typical of lower-cost RISC processors
    - Very low-cost processors may use three stages
    - High-performance processors use 10 or more stages (up to 31!)

Instruction Execution

In RISC machines, all* instructions are executed in the same number of steps.

- Each step is carried out in a separate hardware stage
- We will look at a five-stage design typical of lower-cost RISC processors
    - Very low-cost processors may use three stages
    - High-performance processors use 10 or more stages (up to 31!)

* except for the exceptions, *e.g.*, LDM, STM, division, etc

Load Instruction

```
LDR R5, [R7, R8]  // R5 <-- Mem[R7 + R8]
```

1. Fetch the instruction and increment the program counter
   `IR` ← `Mem[PC]`; `PC` ← `PC + 4`
2. Decode the instruction and read registers R7 and R8 from the register file (RF)
3. Compute the effective address: `Z` ← `R7 + R8`
4. Read the memory source operand: `Y` ← `Mem[Z]`
5. Write into the *destination register*: `R5` ← `Y`

ALU Instruction

```
ADD R3, R4, R5    // R3 <-- R4 + R5
```

1. Fetch the instruction and increment the program counter
   `IR` ← `Mem[PC]`; `PC` ← `PC + 4`
2. Decode the instruction and read registers R4 and R5 from the register file (RF)
3. Compute the sum: `Z` ← `R4 + R5`
4. Take no action except: `Y` ← `Z`
5. Write into the *destination register*: `R3` ← `Y`

The CPU does nothing in the memory access stage, but this step must occur regardless so that all instructions take the same number of steps.

ALU Instruction with Immediate Operand

```
ADD R3, R4, #1000    // R3 <-- R4 + 1000
```

The immediate operand is put in the instruction by the assembly, and can be found in the IR.

1. Fetch the instruction and increment the program counter
   $\texttt{IR} \leftarrow \texttt{Mem[PC]}$; $\texttt{PC} \leftarrow \texttt{PC + 4}$
2. Decode the instruction and read register R4 from the register file
3. Compute the sum: $\texttt{Z} \leftarrow \texttt{R4 + 1000}$
4. Take no action except: $\texttt{Y} \leftarrow \texttt{Z}$
5. Write into the *destination register*: $\texttt{R3} \leftarrow \texttt{Y}$

Store Instruction with Immediate Operand

```
STR R6, [R8, #1000]  // Mem[R8+1000] <-- R6
```

1. Fetch the instruction and increment the program counter
   `IR` ← `Mem[PC]`; `PC` ← `PC + 4`
2. Decode the instruction and read register `R6` and `R8` from the register file (RF)
3. Compute the effective address: `Z` ← `R8 + 1000`;
   Hold onto our data: `M` ← `R6`
4. Write the contents of `R6` to the memory location `R8 + 1000`:
   `Mem[Z]` ← `M`
5. Take no action

The CPU does nothing in the register-write stage, but this step must occur regardless.

Summary of Instruction Execution Stages

Typical RISC CPUs break instruction execution into the following steps:

1. Fetch an instruction into the `IR`; increment `PC`
2. Decode the instruction; read register operands from the RF
3. Execute an ALU operation
4. Access data Memory if the instruction is a load or store
5. Write back the result to the destination register in the RF

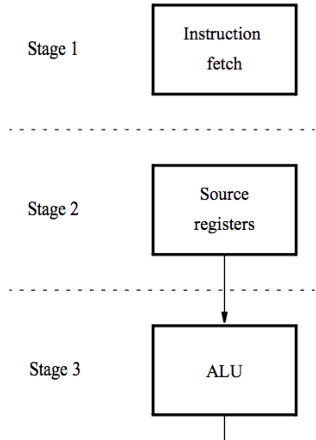
Often each step is performed by a different hardware stage; such a processor uses five stages.

- Stages 1, 2, and 3 will be used for all instructions
- Stages 4 and 5 only perform useful work for a subset of instructions

5-stage RISC Processor

Instruction processing moves from stage to stage with each clock cycle.

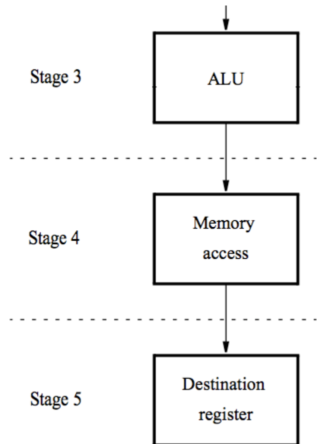
- Stage 1, Fetch: the instruction is read from memory
- Stage 2, Decode: the instruction is decoded, and source registers are read
- Stage 3, Execute: ALU operations are performed



5-stage RISC Processor

Instruction processing moves from stage to stage with each clock cycle.

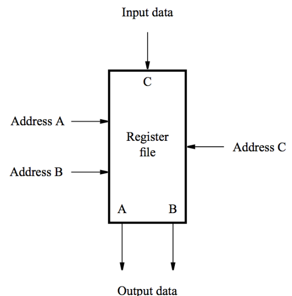
- Stage 4, Memory: memory is read or written, if applicable
- Stage 5, Write back: the result of the instruction is written to the destination register, if applicable



Register File

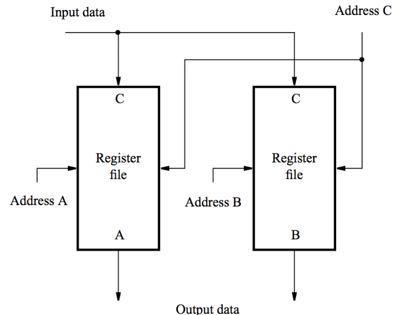
The register file needs to be able to perform two simultaneous reads.
How can we read more than one thing out of a memory at a time?

- A dual-ported SRAM is one way
- Inside, two sets of word lines make it possible to select more than one row at a time (using two decoders)
- Two sets of bit lines make it possible to produce two results at the same time, too
- This requires more transistors (*e.g.*, 8), and extra wiring: higher area (cost), and higher power



Register File

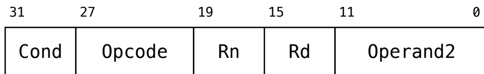
- Using two single-ported SRAMs is an alternative
- Each SRAM has the same data for each register
- One operand is read out of each in parallel
- Each SRAM is updated on each write to the RF
- Is this a good trade-off? Depends! SRAM design is *hard*.



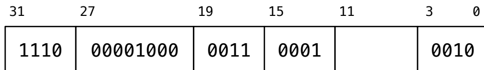
Reading the Register File

How do we decode an instruction and read the RF at the same time?

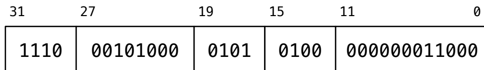
In RISC ISAs, the register fields are always in the same bit positions.




```
ADD R1, R3, R2     // R1 <-- R3 + R2
```

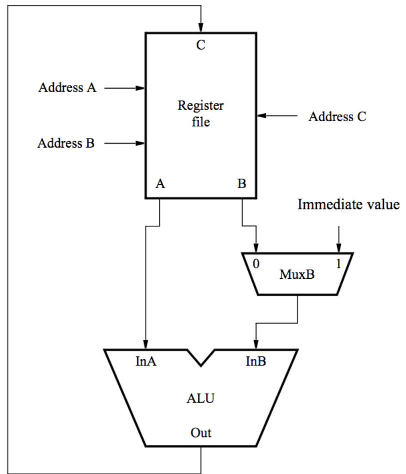



```
ADD R4, R5, #24    // R4 <-- R5 + #24
```



Arithmetic and Logic Unit

- Both source operands and the destination location are in the register file
- Conceptually, the system functions as if the output of the ALU is connected to the input of the RF
- One operand (**RA**) always comes from the RF
- The second operand may come from the RF (**RB**) or the `IR` (immediate value)



There is always an address B, and **RB** output of the RF, but control signals determine what inputs the ALU uses.

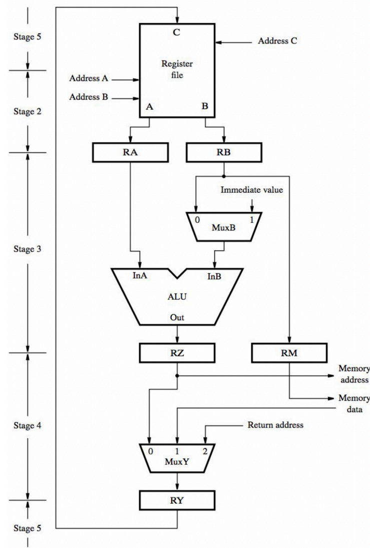
Waiting for Memory

We have assumed so far that all memory accesses take one clock cycle. Is this realistic?

- If we're using a cache, yes, *on average*
- On a cache miss, the processor waits
  - Miss in L1, but hit in L2? A short wait.
  - Have to access main memory? A long wait.
- The memory interface generates a signal called memory function completed (MFC)
- The processor extends the duration of the memory step (in clock cycles) until MFC is asserted

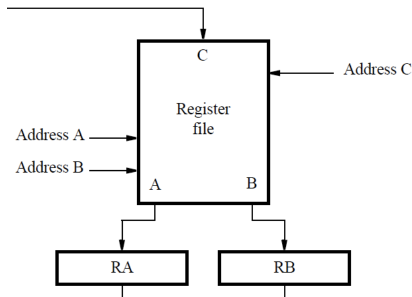
The Datapath: Stages 2-5

- Inter-stage registers RA, RA, RZ, RM, and RY are used to carry data from one stage to the next
- Register file: used in stages 2 and 5; first, to read operands
- ALU: used in stage 3
- Memory: used in stage 4
- Write-back: the final stage is used to write the result to the register file



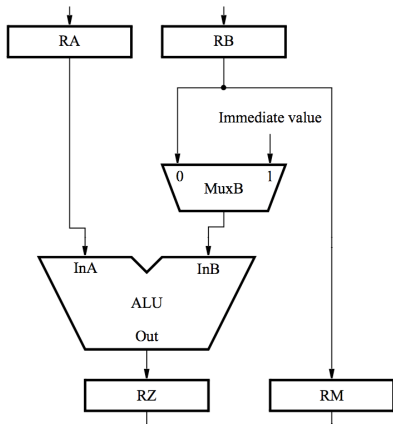
Register File: Stages 2 and 5

- Address inputs are connected to the corresponding bit fields in the `IR`
- Source registers are read (in the first half of) stage 2, and saved in RA and RB
- In (the second half of) stage 5, the result is stored in the destination register selected by `Address C` (`IR` bits)



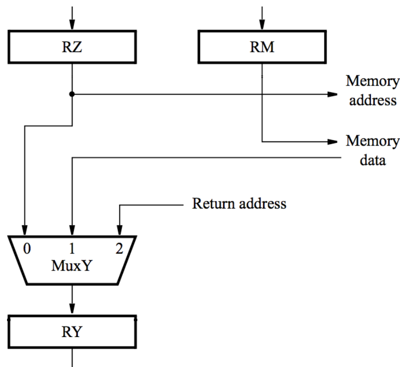
ALU Stage

- The ALU performs (at least part of) the operation specified by the instruction
- Multiplexer `MuxB` selects either `RB` or the immediate field of `IR`
- The result is stored in `RZ`
- Data to be written to memory is transferred from `RB` to `RM`



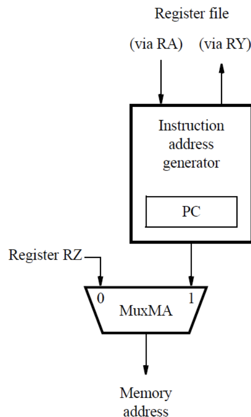
Memory Stage

- Load and store instructions use RZ to address memory
- RM is used to send data to memory on a store
- MuxY selects memory data on a load
- The result from the previous stage (0), or data from memory (1), or return address (2), go in RY



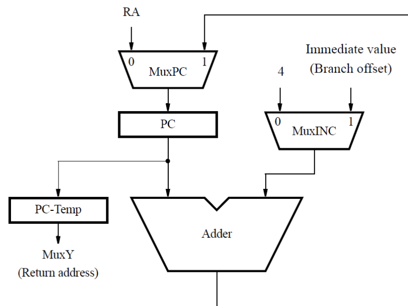
Memory Address Generation

- `MuxMA` selects `PC` when fetching instructions
- The instruction address generator increments `PC` after fetching an instruction, adjusts `PC` in response to branches, and returns values for `LR` on subroutine calls
- `MuxMA` selects `RZ` when reading/writing data operands



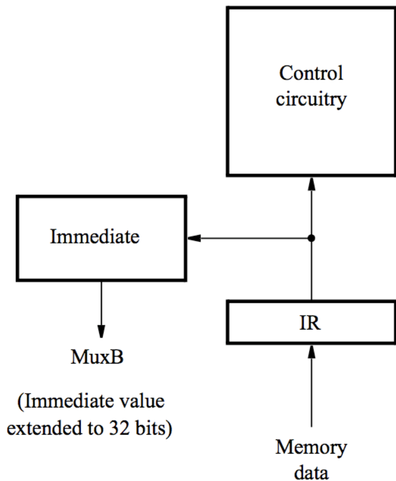
Instruction Address Generator

- Connections to RY and RA support subroutine calls and returns respectively
  - On subroutine call, LR $\leftarrow$ PC
  - On return, PC $\leftarrow$ LR
- Immediate value from `IR` is (sign) extended prior to addition



Processor Control

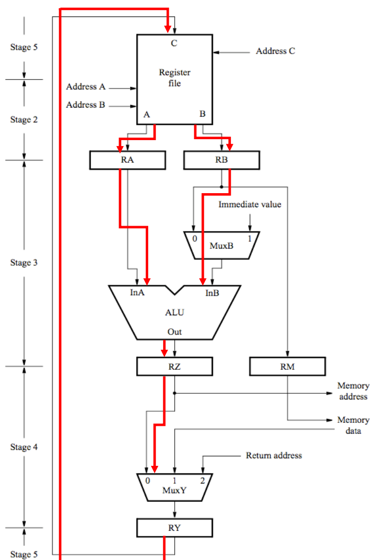
- When an instruction is read, it is saved in `IR`
- Control circuitry (an FSM) decodes the instruction, generating appropriate mux control signals, etc
- The Immediate block extends the immediate field of the instruction to 32 bits based on the instruction type: arithmetic, *sign-extended*; logic, *zero-padded*



Example: ADD

```
ADD R3, R4, R5     // R3 <-- R4 + R5
```

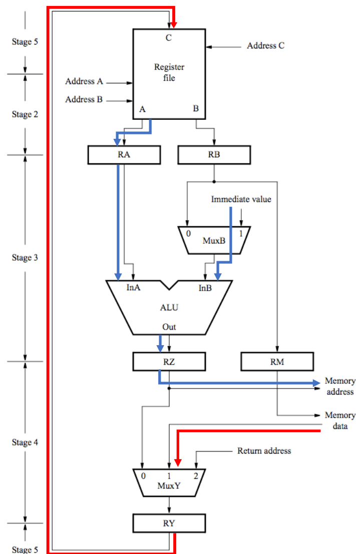
1. Memory address ← PC,
   Read memory,
   IR ← Memory data,
   PC ← PC + 4
2. Decode instruction
   RA ← R4
   RB ← R5
3. RZ ← RA + RB
4. RY ← RZ
5. R3 ← RY



Example: LDR

```
LDR R5, [R7, #24] // R5 <-- Mem[R7 + 24]
```

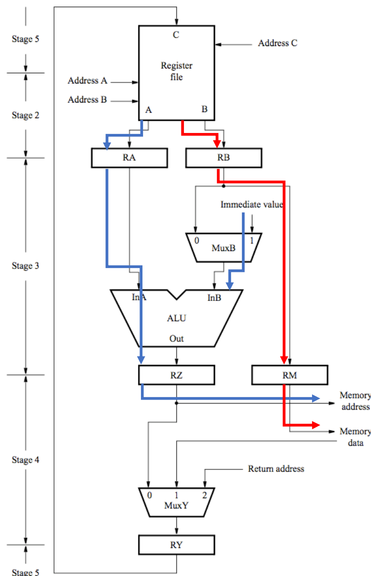
1. Memory address ← PC,
   Read memory,
   IR ← Memory data,
   PC ← PC + 4
2. Decode instruction,
   RA ← R7
3. RZ ← RA + 24
4. Memory address ← RZ,
   Read memory,
   RY ← Memory data
5. R5 ← RY



Example: STR

```
STR R6, [R8, #24] // Mem[R8 + 24] <-- R6
```

1. `Memory address` ← `PC`,
   Read memory,
   `IR` ← `Memory data`,
   `PC` ← `PC + 4`
2. Decode instruction,
   `RA` ← `R8`,
   `RB` ← `R6`
3. `RZ` ← `RA + 24`
   `RM` ← `RB`
4. `Memory address` ← `RZ`,
   `Memory data` ← `RM`,
   Write memory
5. No action taken



Example: B

```
B   label          // PC <-- PC + displacement
```

1. `Memory address` $\leftarrow$ `PC`,
   Read memory,
   `IR` $\leftarrow$ `Memory data`,
   `PC` $\leftarrow$ `PC + 4`
2. Decode instruction
3. `PC` $\leftarrow$ `PC + displacement`
4. No action taken
5. No action taken

Example: BEQ

Assume a RISC processor that *does not* employ conditional execution. Conditional branches make comparisons.

```
BEQ R5, R6, label // If R5 == R6, PC <-- PC + displacement
```

1. `Memory address` ← `PC`,
   Read memory,
   `IR` ← `Memory data`,
   `PC` ← `PC + 4`
2. Decode instruction,
   `RA` ← `R5`,
   `RB` ← `R6`
3. `RZ` ← `RA - RB`,
   If ALU_{iszero} `== 1` then `PC` ← `PC + displacement`
4. No action taken
5. No action taken

ALU signals when result is zero, positive, and negative, and when overflow, and carry out occur.

Note that the ARM ISA decouples the condition from the branch:

- An instruction update the CPSR (ALU signals flow into CPSR)
- A conditional instruction executes only if condition is true (by checking CPSR)

Control Design

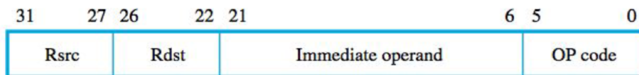
Textbook§5.5-5.7

Example RISC Instruction Format

The instruction register holds the current instruction; different groups of bits (fields) in this register are used to determine what the instruction, is and therefore what control signals to activate.

Consider the following instruction encoding schemes:

(a) Register-operand format

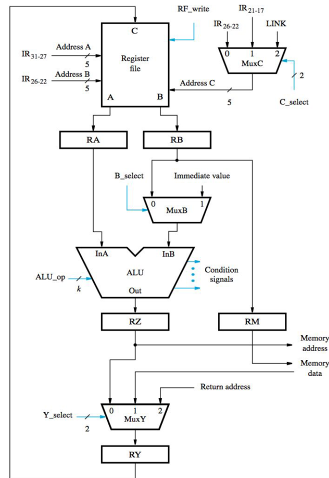


(b) Immediate-operand format

Control Signals

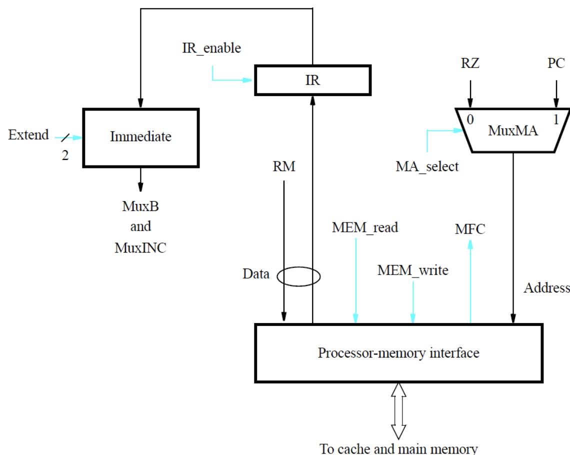
Control signals are (a) directly derived from `IR` fields, or (b) determined based on the OP code.

1. `IR` fields select registers from the RF
2. Mux inputs are chosen to direct flow of register outputs
3. `ALU_op` determines the function of the ALU; ALU also generates signals
4. An FSM coordinates when `PC`, `IR`, RF, and memory are written
5. Interstage registers are always enabled; their contents only matter when the stage they drive is active



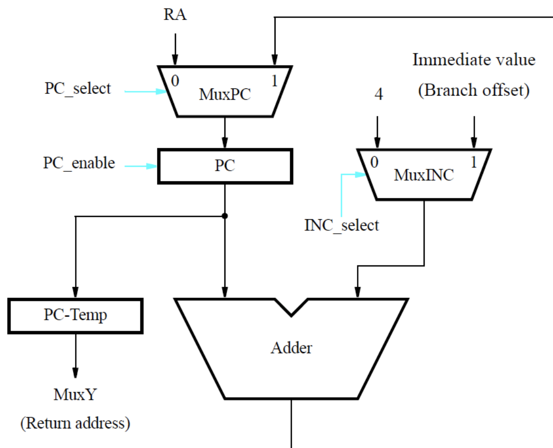
Memory and IR Control Signals

- `IR` is written only after `MFC` is asserted
- This design can sign-extend 16 bits, zero pad 16 bits, or prepare a 26-bit immediate for use in subroutine calls



Instruction Address Generator Control Signals

Control signals determine the source used to update PC, and when.



Control Signal Generation

The control unit generates the control signals so the actions in the datapath take place in the correct sequence and at the correct time.

- Two basic approaches:
  - Hardwired control (typical of RISC, *e.g.*, ARM)
  - Microprogramming (typical of CISC, *e.g.*, Intel)
- Hardwired control involves implementing an FSM
- The FSM keeps count of the current stage: one cycle each, except for memory, which may take longer
- FSM inputs: `IR`, ALU output signals, external inputs (*e.g.*, interrupts)
- FSM outputs: control signals

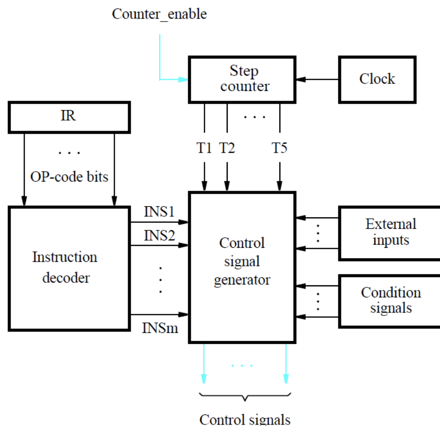
Hardwired Control Signal Generation

Example: stage 1 (fetch)

when T1 == 1:

```
MA_select ← 1
MEM_read ← 1
IR_enable ← MFC

INC_select ← 0
PC_select ← 1
PC_enable ← MFC
```



Example: Counter_enable

What is the logic expression for `Counter_enable`?

- Control must wait until `MFC` to be asserted before incrementing the step counter in a step in which `MEM_read` or `MEM_write` command is asserted
- `Counter_enable` should be asserted in any step in which `WFMC` (wait for memory to complete) is not asserted: *e.g.*,`T1 + T4`
- Otherwise, it should be asserted when `MFC` is asserted

$$\texttt{Counter_enable} \leftarrow \overline{\texttt{WFMC}} + \texttt{MFC}$$

Example: PC_enable

What is the logic expression for the `PC_enable`?

- Control must make sure PC is incremented only once when a execution step is extended for more than one clock cycle
- Writing to PC should only be enabled when (a) `MFC` is asserted, or (b) in stage 3 of branch instructions

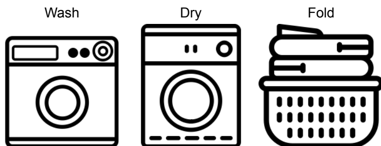
$$\texttt{PC_enable} \leftarrow \texttt{T1} \cdot \texttt{MFC} + \texttt{T3} \cdot \texttt{BR}$$

Pipelining

Textbook§6.1-6.7

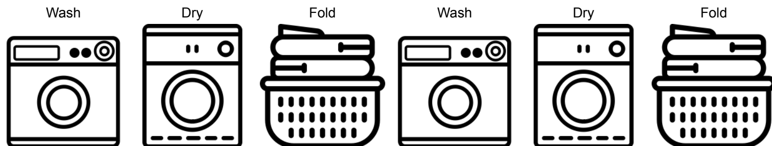
Example

Consider doing laundry. If each operation requires one hour, the latency per load is three hours.



Example

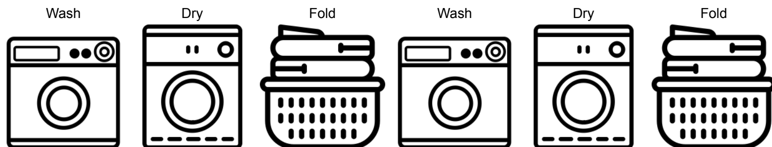
Consider doing laundry. If each operation requires one hour, the latency per load is three hours.



Two loads? Six hours total.

Example

Consider doing laundry. If each operation requires one hour, the latency per load is three hours.

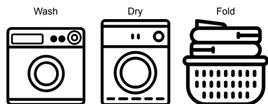


Two loads? Six hours total.

This is inefficient when there's a lot of laundry: when the dryer is working, the washer is idle!

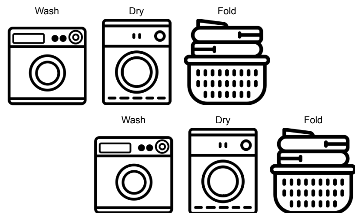
Example

What happens if we make use of washer and dryer simultaneously on different loads?



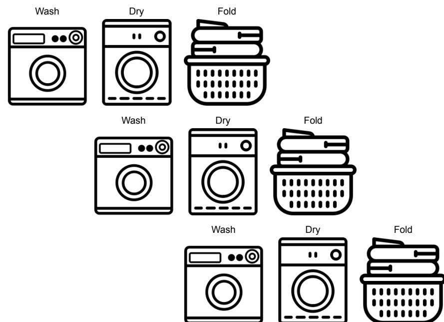
Example

What happens if we make use of washer and dryer simultaneously on different loads?



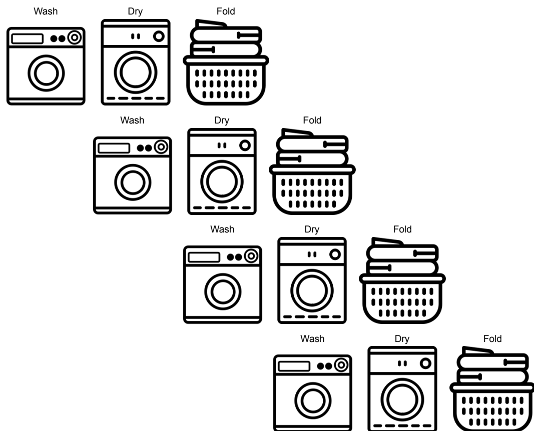
Example

What happens if we make use of washer and dryer simultaneously on different loads?



Example

What happens if we make use of washer and dryer simultaneously on different loads?



Six hours, with pipelining? *Four* loads, instead of two.

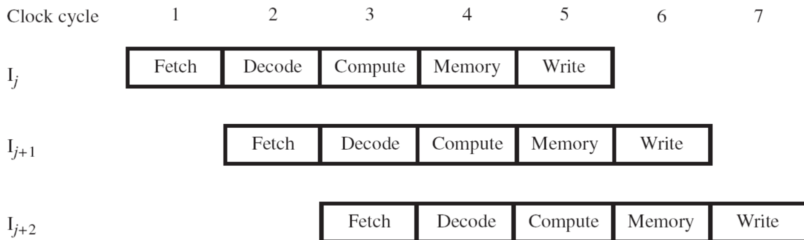
What is pipelining?

Pipelining is applying the “assembly line” concept to the execution of instructions

- Instruction execution is divided into distinct steps (like we’ve already done)
- Multiple instructions are executed simultaneously by overlapping the steps of different instructions:
  - Only one instruction is started at a time
  - Each hardware stage is working on a different instruction
  - This keeps all stages busy, dramatically improving performance

Ideal Pipelining

In the ideal case, a new instruction is started each clock cycle, and each instruction only takes a single cycle in each step.



What are some reasons why this ideal may not be always achievable?

Pipeline Organization

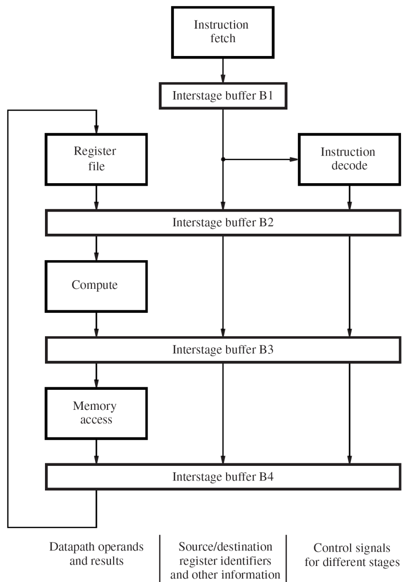
- Use `PC` to fetch a new instruction every* cycle
- Instruction-specific information moves with instructions through the different stages
- Interstage buffers (pipeline registers) hold this information, incorporating `RA`, `RB`, `RM`, `RY`, `RZ`, `IR`, and `PC-Temp` registers
- The buffers also hold control signals: *e.g.*, mux inputs are determined during decode, but applied when appropriate

Pipeline Organization

- Use PC to fetch a new instruction every* cycle
- Instruction-specific information moves with instructions through the different stages
- Interstage buffers (pipeline registers) hold this information, incorporating RA, RB, RM, RY, RZ, IR, and PC-Temp registers
- The buffers also hold control signals: *e.g.*, mux inputs are determined during decode, but applied when appropriate

* Except when something prevents an instruction from advancing!

Pipeline Organization



What can stall the pipeline?

Instructions advance, one stage per cycle, unless something occurs to stall an instruction. Circumstances in which one instruction causes a delay in another instruction are called hazards, and they come in three flavors.

- Structural hazards: caused by contention for a shared resource (*e.g.*, memory)
- Data hazards: occur when one instruction must wait for the result of another
- Control hazards: caused by branch instructions delaying instruction fetch

Instructions may also be delayed when our assumption that each stage takes a single cycle is violated (*e.g.*, when a memory access results in a cache miss).

Data Dependencies

Consider the following assembly.

```
ADD   R2, R3, R7  // R2 <-- R3 + R7
SUB   R9, R2, R8  // R9 <-- R2 - R8
```

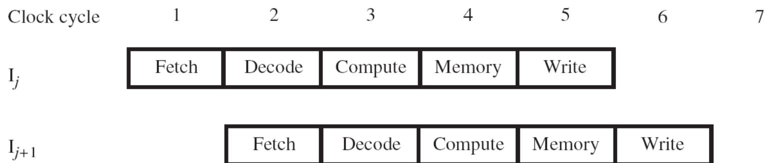
R2 is the (a) destination of the add instruction, and (b) source for the subtract instruction.

- There is a data dependency between ADD and SUB: SUB cannot be executed until we have the result of the ADD.
- With no pipelining, there's no problem: the result is in R2 because ADD completes before SUB begins.
- With pipelining, SUB starts before ADD finishes.

Data Hazards

```
ADD   R2, R3, R7   // R2 <-- R3 + R7
SUB   R9, R2, R8   // R9 <-- R2 - R8
```

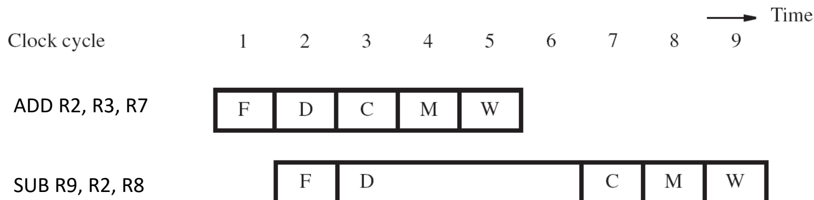
Suppose ADD is instruction I_j and SUB is instruction I_{j+1}:



- I_{j+1} reads its operands in cycle 3
- But the result of I_j is written in cycle 5 (to be read in cycle 6)
- I_j and I_{j+1} cannot execute simultaneously because of the data dependency
- This is a data hazard

To resolve this, we delay SUB until its operands are available.

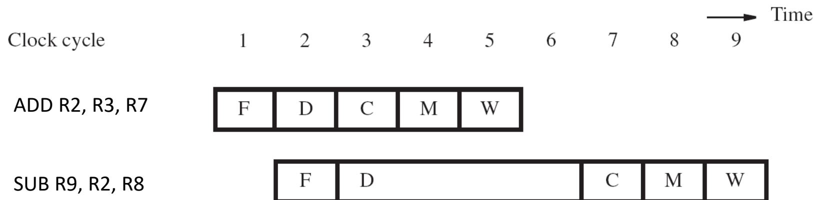
Stalling the Pipeline



We must delay the SUB instruction until it can read the result of the ADD from R2.

- R2 is written in cycle 5
- R2 can be read in cycle 6
- The CPU discovers the dependency during decode in cycle 3
- SUB stalls in decode for three cycles (3, 4, 5) before reading R2 in cycle 6

Stalling the Pipeline



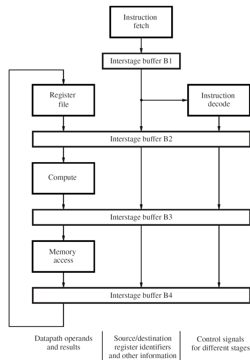
Control circuitry detects the dependencies during decode.

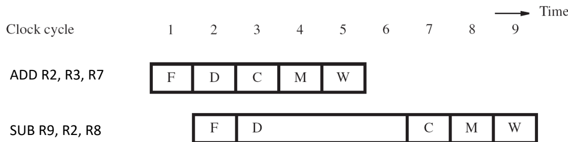
- Interstage buffers carry register identifiers for source(s) and destination of instructions
- In cycle 3, control compares the destination register in Compute (R2) against source(s) in Decode (R2 and R8)
- In this case, R2 matches; SUB is kept in Decode while ADD is allowed to continue

Stalling the Pipeline

What happens when ADD leaves Compute and enters Memory?

- B1 is not clocked, holding SUB in decode
- Meanwhile, control signals in Compute are set to create an *implicit* NOP (no-operation)
- These NOPs (also called *bubbles*) propagate through the pipeline
- Then, Control compares sources in Decode and destinations in later stages
- The dependency remains (ADD in Memory); SUB is stalled again (B1 not clocked)
- This repeats until the dependency clears

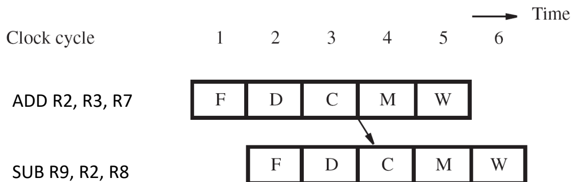




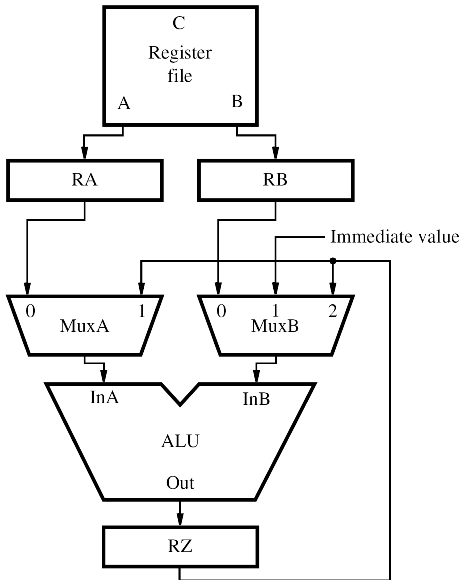
Can we avoid stalling?

We can avoid some hazards by adding extra hardware to the pipeline, and more complex logic to the control circuitry.

- Operand forwarding handles some data dependencies without stalling the pipeline
- In our example, ADD's result is in RZ (within B3) in cycle 4
- We can add inputs to our ALU operand muxes and forward the result from stage 4 to stage 3



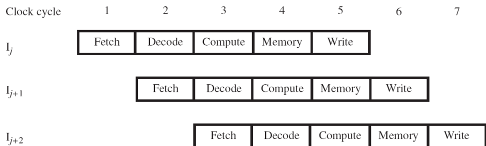
Forwarding: Memory to Compute

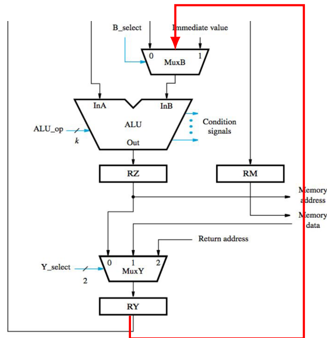


Forwarding: Write-back to Compute

If an instruction separates two with a dependency, we still must stall if we cannot forward. Solution: add more forwarding paths!

```
ADD   R2, R3, R7   // R2 <-- R3 + R7
ORR   R4, R5, R6   // R4 <-- R5 || R6
SUB   R9, R2, R8   // R9 <-- R2 - R8
```



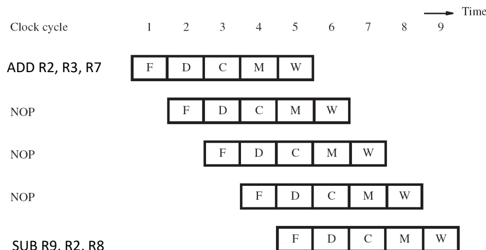


Mux inputs must also be added to accept forwarding from Write-back.

Handling Dependencies in Software

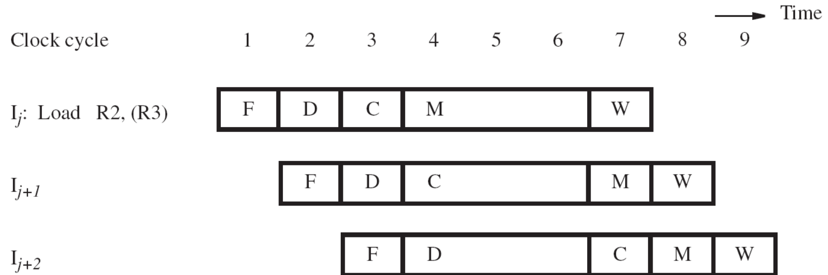
Data dependencies are evident during assembly, and can therefore be handled in software (if, *e.g.*, we do not intend to detect or mitigate them in hardware).

- The assembler inserts three *explicit* NOP instructions
- SUB does not enter decode until the result of ADD is availble
- The assembler can optimize, replacing NOP with independent instructions



Memory Delays

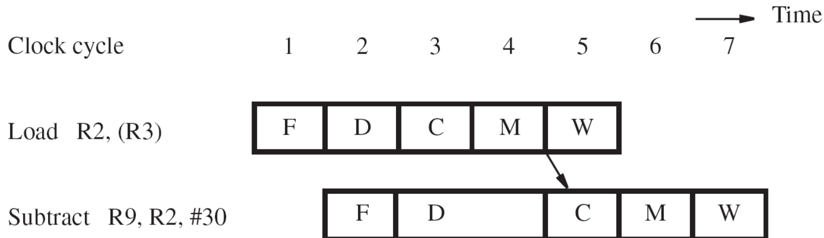
Cache misses can delay instructions in either the Fetch or Memory stages, *e.g.*,



Memory Delays

Even when a load hits in cache, there may be delay due to a data dependency.

- A one-cycle stall is required before the result can be forwarded from the Write-back stage
- Optimize by inserting a useful instruction between the two

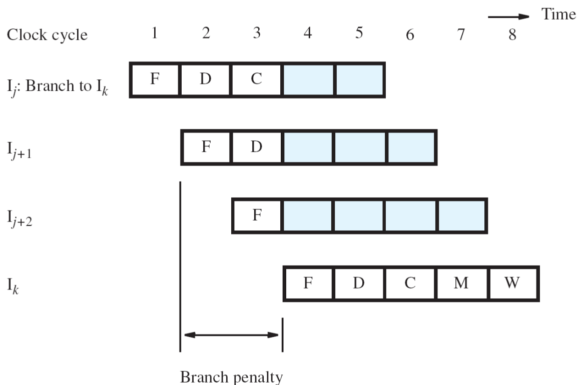


Control Hazards

Remember that ideal pipelining expects that we can fetch a new instruction each cycle, while the previous instruction is decoded.

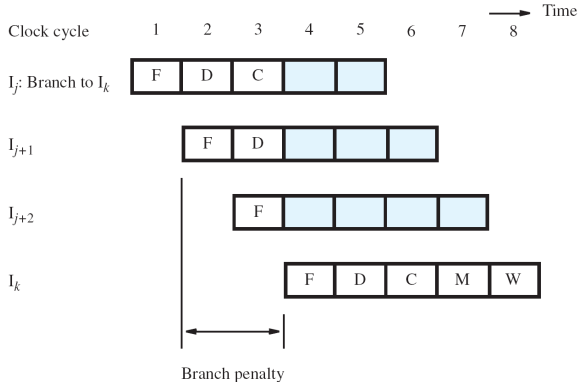
- Branch instructions must (a) compute the target address, and (b) potentially compare registers
- This comparison determines whether to go to the target address, or execute the fall-through instruction
- A hazard occurs because these operations occur in later stages (*e.g.*, Compute)

Unconditional Branches



- Target address $(\mathit{offset} + (PC + 4))$ is computed in cycle 3
- Meanwhile, fetch in cycles 2 $(PC + 4)$ and 3 $((PC + 4) + 4)$
  Mystery solved! PC is 8 ahead!
- These instructions are discarded, resulting in a 2 cycle penalty

Reducing the Branch Penalty



We can reduce the branch penalty by computing the target earlier.

- Add an adder to the decode stage
- This shortens the branch penalty by one cycle

We are adding HW (*i.e.*, cost and energy) to improve performance.

Conditional Branches

```
BEQ R5, R6, label // If R5 == R6, PC <-- PC + displacement
```

- Conditional branches must compute the target address and compare registers
- We can compute the target in Decode with an extra adder
- We can also make a comparison in Decode with an extra comparator

We are adding hardware *again* to improve performance.

What’s Next?

This set of lectures introduced the basics of processor implementation. We’ve looked at:

- Data path elements and design
- Control circuitry design
- Pipelining and pipeline hazards

Next we’ll look at computer hardware for arithmetic.