F28HS Hardware-Software Interface: Systems Programming

Hans-Wolfgang Loidl

School of Mathematical and Computer Sciences,
Heriot-Watt University, Edinburgh

Semester 2 — 2021/22

[0] No proprietary software has been used in producing these slides

Outline

1. Lecture 1: Introduction to Systems Programming
2. Lecture 2: Systems Programming with the Raspberry Pi
3. Lecture 3: Memory Hierarchy
   - Memory Hierarchy
     - Principles of Caches
4. ~~Lecture 4: Programming external devices~~
   - ~~Basics of device-level programming~~
5. Lecture 5: Exceptional Control Flow
6. Lecture 6: Computer Architecture
   - Processor Architectures Overview
     - Pipelining
7. ~~Lecture 7: Code Security: Buffer Overflow Attacks~~
8. ~~Lecture 8: Interrupt Handling~~
9. Lecture 9: Miscellaneous Topics
10. Lecture 10: Revision

HERIOT WATT UNIVERSITY

Lecture 6:
Computer Architecture

Classes of Computer Architectures

- There is a wide range of computer architectures from small-scale (embedded) to large-scale (super-computers)
- In this course we focus on **embedded systems**
- A key requirement for these devices is **low power consumption**
- This is also increasingly important for main-stream hardware and even for super-computing
- Embedded devices are found in cars, planes, house-hold devices, network-devices, cell-phones etc
- This is the most rapidly growing market for computer hardware

Number of processors produced

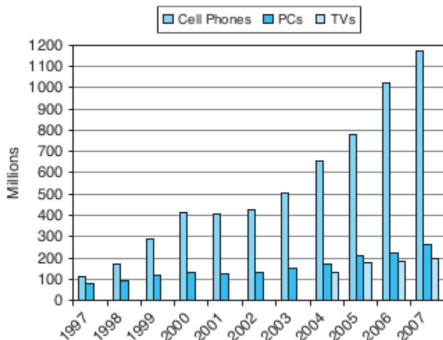


FIGURE 1.1 The number of cell phones, personal computers, and televisions manufactured per year between 1997 and 2007. (We have television data only from 2004.) More than a billion new cell phones were shipped in 2006. Cell phones sales exceeded PCs by only a factor of 1.4 in 1997, but the ratio grew to 4.5 in 2007. The total number in use in 2004 is estimated to be about 2.0B televisions, 1.8B cell phones, and 0.8B PCs. As the world population was about 6.4B in 2004, there were approximately one PC, 2.2 cell phones, and 2.5 televisions for every eight people on the planet. A 2006 survey of U.S. families found that they owned on average 12 gadgets, including three TVs, 2 PCs, and other devices such as game consoles, MP3 players, and cell phones.

[0] From Patterson & Hennessy, Chapter 1

Limitations to further improvements

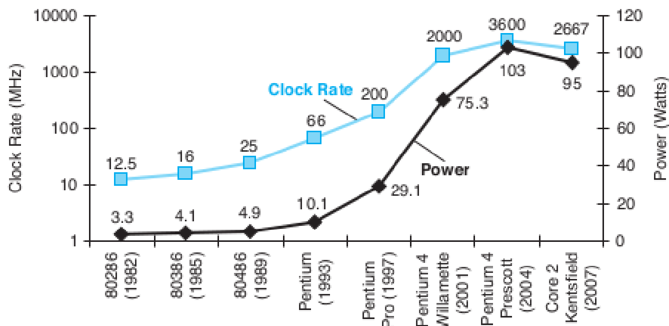


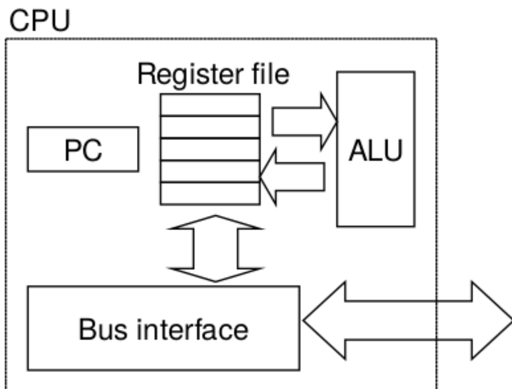
FIGURE 1.15 Clock rate and Power for Intel x86 microprocessors over eight generations and 25 years. The Pentium 4 made a dramatic jump in clock rate and power but less so in performance. The Prescott thermal problems led to the abandonment of the Pentium 4 line. The Core 2 line reverts to a simpler pipeline with lower clock rates and multiple processors per chip.

[0] From Patterson & Hennessy, Chapter 1

Processor Architectures: Introduction

- In this part we take a brief look at the design of processor hardware.
- This view will give you a better understanding of how computers work.
- In particular you will gain a better understanding of issues relevant to **resource consumption.**
- So far we have used a very simple model of a CPU: each instruction is fetched and executed to completion before the next one begins.
- Modern processor architectures use **pipeling** to execute multiple instructions simultaneously (“super-scalar architectures”).
- Special measures need to be taken to ensure that the processor computes the same results as it would with sequential execution.

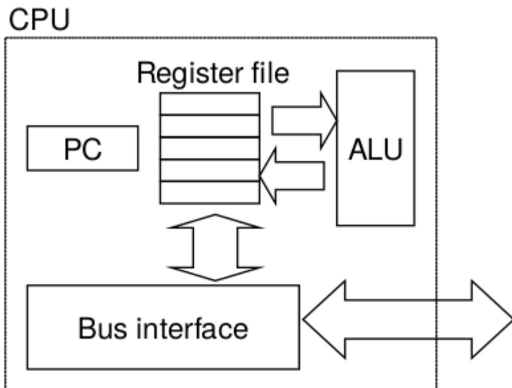
A simple picture of the CPU



- The ALU executes arithmetic/logic operations with arguments in registers
- Load and store instructions move data between memory and registers

HERIOT WATT UNIVERSITY

A simple picture of the CPU



- The ALU executes arithmetic/logic operations with arguments in registers
- Load and store instructions move data between memory and registers

Why should you learn about architecture design?

- It is intellectually interesting and important.
- Understanding how the processor works aids in understanding how the overall computer system works.
- Although few people design processors, many design hardware systems that contain processors.
- You just might work on a processor design.

Stages of executing an assembler instruction

Processing an assembler instruction involves a number of operations:

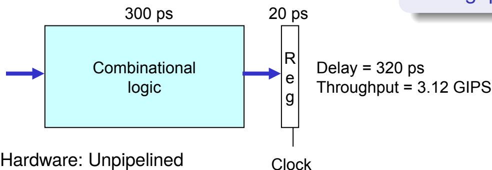
1. **Fetch:** The fetch stage reads the bytes of an instruction from memory, using the program counter (PC) as the memory address.
2. **Decode:** The decode stage reads up to two operands from the register file.
3. **Execute:** In the execute stage, the arithmetic/logic unit (ALU) either performs the operation specified by the instruction, computes the effective address of a memory reference, or increments or decrements the stack pointer.
4. **Memory:** The memory stage may write data to memory, or it may read data from memory.
5. **Write back:** The write-back stage writes up to two results to the register file.
6. **PC update:** The PC is set to the address of the next instruction.

NB: The processing depends on the instruction, and certain stages may not be used.

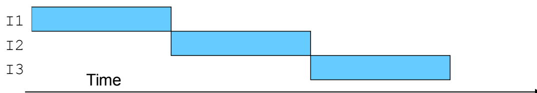
Unpipelined computation hardware

Unpipelined Design

Delay: **320 ps**
Throughput: **3.12 GIPS**



(a) Hardware: Unpipelined



(b) Pipeline diagram

On each 320 ps cycle, the system spends 300 ps evaluating a combinational logic function and 20 ps storing the results in an output register.

[0] From Bryant, Chapter 4

Instruction-level parallelism

- **Key observation:** We can do the different stages of the execution in parallel ("instruction-level parallelism")
- An architecture that allows this kind of parallelism is called **"pipelined"** architecture
- This is a big performance boost: ideally each instruction takes just 1 cycle (as opposed to 5 cycles for the 5 stages of the execution)
- However, the ideal case is often not reached, and modern architecture play clever tricks to get closer to the ideal case: branch prediction, out-of-order execution etc

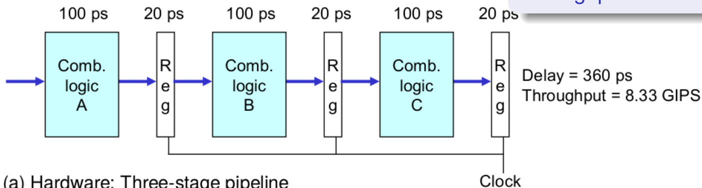
HERIOT WATT UNIVERSITY

Three-stage pipelined computation hardware

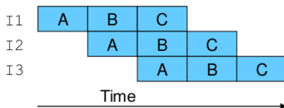
Three-stage Pipeline

Delay: **360 ps**

Throughput: **8.33 GIPS**



(a) Hardware: Three-stage pipeline

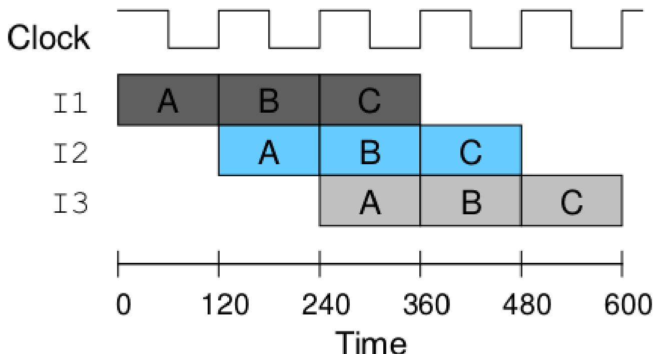


(b) Pipeline diagram

The computation is split into stages A, B, and C. On each 120-ps cycle, each instruction progresses through one stage.

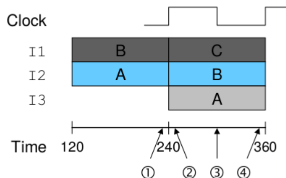
[0] From Bryant, Chapter 4

Three-stage pipeline timing



The rising edge of the clock signal controls the movement of instructions from one pipeline stage to the next.

Example: One clock cycle of pipeline operation.

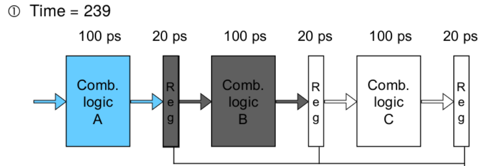


- We now take a closer look on how values are propagated through the pipeline.
- Instruction `I1` has completed stage B
- Instruction `I2` has completed stage A

---

[0]From Bryant, Chapter 4, Fig 4.35

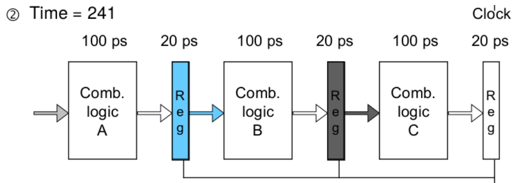
Example: One clock cycle of pipeline operation.



Just **before** clock rise: values have been computed (stage A of instruction I2, stage B of instruction I1), but the pipeline registers have not been updated, yet.

[0] From Bryant, Chapter 4

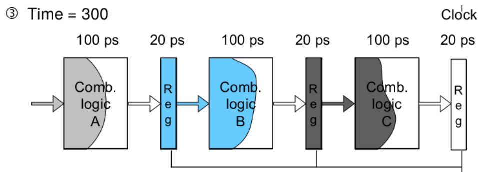
Example: One clock cycle of pipeline operation.



On **clock rise**, inputs are loaded into the pipeline registers.

[0] From Bryant, Chapter 4

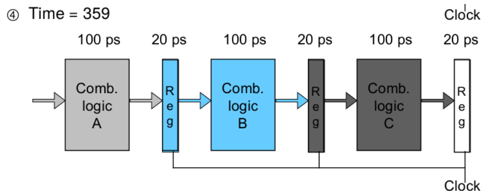
Example: One clock cycle of pipeline operation.



Signals then propagate through the combinational logic (possibly at different rates).

[0] From Bryant, Chapter 4

Example: One clock cycle of pipeline operation.



Before time 360, the result values reach the inputs of the pipeline registers, to be propagated at the next rising clock.

[0] From Bryant, Chapter 4

Multiple-clock-cycle pipeline diagram

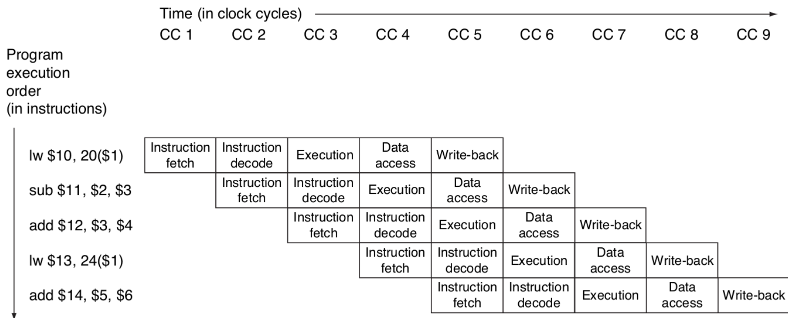
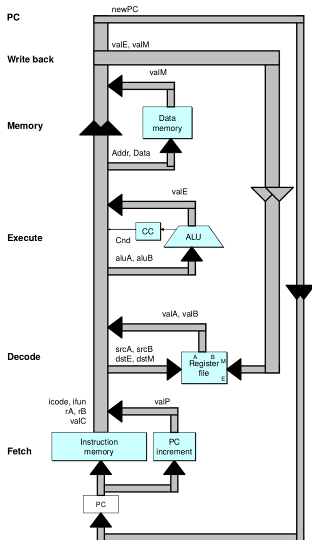


FIGURE 4.44 Traditional multiple-clock-cycle pipeline diagram of five instructions in Figure 4.43.

[0] From Patterson & Hennessy, Chapter 4

Abstract view of a sequential processor



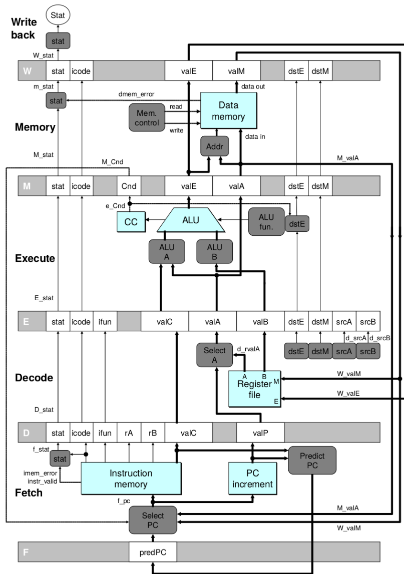
The information processed during execution of an instruction follows a clockwise flow starting with an instruction fetch using the program counter (PC), shown in the lower left-hand corner of the figure.

Discussion of pipelined execution

The main pipeline **stages** are:

- **Fetch:** Using the program counter register as an address, the instruction memory reads the bytes of an instruction. The PC incrementer computes valP, the incremented program counter.
- **Decode:** The register file has two read ports, A and B, via which register values valA and valB are read simultaneously.
- **Execute:** This uses the arithmetic/logic (ALU) unit for different purposes according to the instruction type: integer operations, memory access, or branch instructions.
- **Memory:** The Data Memory unit reads or writes a word of memory (memory instruction). The instruction and data memories access the same memory locations, but for different purposes.
- **Write back:** The register file has two write ports. Port E is used to write values computed by the ALU, while port M is used to write values read from the data memory.

Abstract view of a pipelined processor



Hardware structure of a pipelined implementation. By inserting pipeline registers between the stages, we create a five-stage pipeline.

[0] From Bryant, Chapter 4

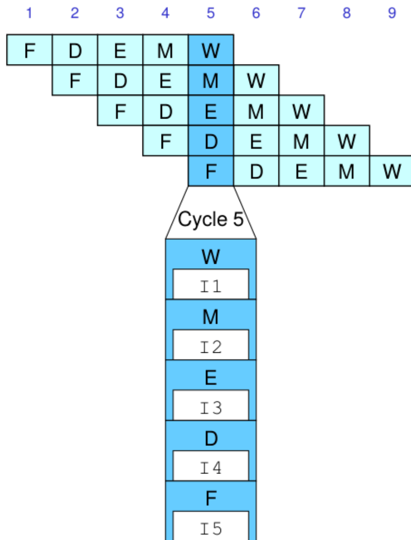
Pipeline registers

The pipeline **registers** are labeled as follows:

- **F** holds a predicted value of the program counter.
- **D** sits between the fetch and decode stages. It holds information about the most recently fetched instruction for processing by the decode stage.
- **E** sits between the decode and execute stages. It holds information about the most recently decoded instruction and the values read from the register file for processing by the execute stage.
- **M** sits between the execute and memory stages. It holds the results of the most recently executed instruction for processing by the memory stage. It also holds information about branch conditions and branch targets for processing conditional jumps.
- **W** sits between the memory stage and the feedback paths that supply the computed results to the register file for writing and the return address to the PC selection logic when completing a return instruction.

Example of instruction flow through pipeline

```
MOV   R1,#20    @I1
MOV   R2,#05    @I2
MUL   R0,R1,R2  @I3
MOV   R7,#00    @I4
SWI   0         @I5
```



[0] From Bryant, Chapter 4

The ARM picture

The pipeline in the BCM2835 SoC for the RPi has 8 pipeline stages:

1. **Fe1:** The first Fetch stage, where the address is sent to memory and an instruction is returned.
2. **Fe2:** Second fetch stage, where the processor tries to predict the destination of a branch.
3. **De:** Decoding the instruction.
4. **Iss:** Register read and instruction issue
5. **Only for ALU operations:**
   1. **Sh:** Perform shift operations as required.
   2. **ALU:** Perform arithmetic/logic operations.
   3. **Sat:** Saturate integer results.
6. **WBi:** Write back of data from any of the above sub-pipelines.

---

[0] See slidesRPiArch and the table in Smith's book

The ARM picture

The pipeline in the BCM2835 SoC for the RPi has 8 pipeline stages:

1. **Fe1:** The first Fetch stage, where the address is sent to memory and an instruction is returned.
2. **Fe2:** Second fetch stage, where the processor tries to predict the destination of a branch.
3. **De:** Decoding the instruction.
4. **Iss:** Register read and instruction issue
5. **Only for Multiply operations:**
   1. **MAC1:** First stage of the multiply-accumulate pipeline.
   2. **MAC2:** Second stage of the multiply-accumulate pipeline.
   3. **MAC3:** Third stage of the multiply-accumulate pipeline.
6. **WBi:** Write back of data from any of the above sub-pipelines.

[0] See slidesRPiArch and the table in Smith's book

The ARM picture

The pipeline in the BCM2835 SoC for the RPi has 8 pipeline stages:

1. **Fe1:** The first Fetch stage, where the address is sent to memory and an instruction is returned.
2. **Fe2:** Second fetch stage, where the processor tries to predict the destination of a branch.
3. **De:** Decoding the instruction.
4. **Iss:** Register read and instruction issue
5. **Only for Load/Store operations:**
   1. **ADD:** Address generation stage.
   2. **DC1:** First stage of data cache access.
   3. **DC2:** Second stage of data cache access.
6. **WBi:** Write back of data from any of the above sub-pipelines.

[0] See slidesRPiArch and the table in Smith's book

Pipelining and branches

- How can a pipelined architecture deal with conditional branches?
- In this case the processor doesn't know the successor instruction until further down the pipeline.
- To deal with this, modern architectures perform some form of **branch prediction** in hardware.
- There are two forms of branch prediction:
  - static branch prediction always takes the same guess (e.g. guess **always taken**)
  - dynamic branch prediction uses the history of the execution to take better guesses
- Performance is significantly higher when branch predictions are correct
- If they are wrong, the processor needs to stall or inject bubbles into the pipeline

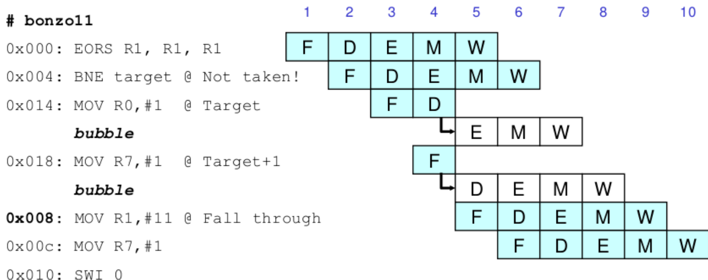
Example: bad branch prediction

```
        .global _start
        .text
_start:  EORS R1, R1, R1   @ always 0
         BNE  target       @ Not taken
         MOV  R0, #11      @ fall through
         MOV  R7, #1
         SWI  0
target:  MOV  R0, #1
         MOV  R7, #1
         SWI  0
```

Branch prediction: we assume the processor takes an **always taken** policy, i.e. it always assumes that that a branch is taken

NB: the conditional branch (BNE) will never be taken, because exclusive-or with itself always gives 0, i.e. this is a deliberately bad example for the branch predictor

Processing mispredicted branch instructions.



- Predicting "branch taken", instruction `0x014` is fetched in cycle 3, and instruction `0x018` is fetched in cycle 4.
- In cycle 4 the branch logic detects that the branch is **not** taken
- It therefore abandons the execution of `0x014` and `0x018` by injecting **bubbles** into the pipeline.
- The result will be as expected, but performance is sub-optimal!

[0] Adapted from Bryant, Figure 4.62

Example of bad branch prediction

Code example: `sumav3_asm`

Hazards of Pipelining

- Pipelining complicates the processing of instructions because of:
  - **Control hazards**, where branches are mis-predicted (as we have seen)
  - **Data hazards**, where data dependencies exist between subsequent instructions
- Several ways exist to solve these problems:
  - To deal with **control hazards**, branch prediction is used and, if necessary, partially executed instructions are abandoned.
  - To deal with **data hazards**, bubbles can be injected to delay the execution of instructions, or data in pipeline registers (but not written back) can be forwarded to other stages in the pipeline.
- A lot of the complexities in modern processors is due to deep pipelining, (possibly dynamic) branch prediction, and forwarding of data

For details on pipelining and data hazards, see Bryant & O'Hallaron, *Computer Systems: A Programmer's View*, Chapter 4 (especially Sec 4.4 and 4.5).

HERIOT WATT UNIVERSITY

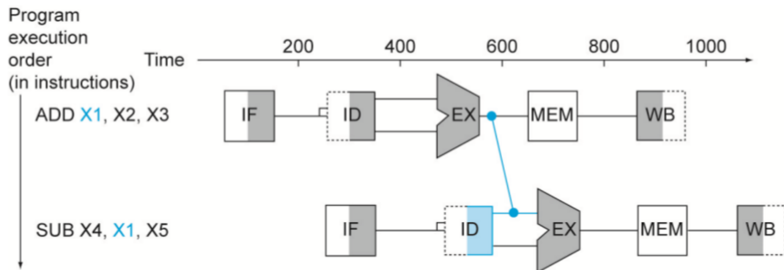
Data Hazards

- The branch-prediction example above was a case of a **control hazard**.
- Now we look into a simple example of a **data hazard**.
- Consider the following simple ARM assembler program:

```
ADD   R3, R1, R2     @  R3 = R1 + R2
SUB   R0, R3, R4     @  R0 = R3 - R4
```

- Note, the result from the first instruction, in `R3`, will only become available in the **write-back** (5th) stage
- But, the data in `R3` is needed already in the **decode** (2nd) stage of the second instruction
- Without intervention, this would stall the pipeline, similar to the branch-mis-prediction case
- The solution to this is to introduce **forwarding** (or by-passing) to the hardware of the processor

A Graphical Representation of Forwarding



[0] From Patterson & Hennessy, Chapter 4

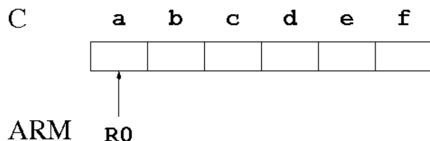
Example: Reordering Code to Avoid Pipeline Stalls

- We have previously examined, how C expressions are compiled to Assembler code. For example, consider this C program fragment:

```
int a, b, c, d, e, f;
a = b + e;
c = b + f;
```

- Knowing about control and data hazards motivates **reordering of code** that should be done by the compiler to avoid pipeline stalls.
- Such reordering is commonly done in the backend of compilers.
- Therefore, the sequence of Assembler instructions might be different from the one you expect.

Data layout and code for a C expression




```
a = b + e

LDR  R1, [R0, #4]
LDR  R2, [R0, #16]
ADD  R3, R1, R2
STR  R3, [R0, #0]
```

[0] From Patterson & Hennessy, Chapter 4

Example: Reordering Code to Avoid Pipeline Stalls

Example: Translate the following C expression into Assembler:

```
int a, b, c, d, e, f;
a = b + e;
c = b + f;
```

Example: We assume the variables are stored in memory, starting from the location held in register `R0`. Here is the naive Assembler code:

```
LDR  R1, [R0, #4]    @ load b
LDR  R2, [R0, #16]   @ load e
ADD  R3, R1, R2      @ b + e
STR  R3, [R0, #0]    @ store a
LDR  R4, [R0, #20]   @ load f
ADD  R5, R1, R4      @ b + f
STR  R5, [R0, #12]   @ store c
```

Can you spot the data hazard in this example?

Example: Reordering Code to Avoid Pipeline Stalls

Example: Translate the following C expression into Assembler:

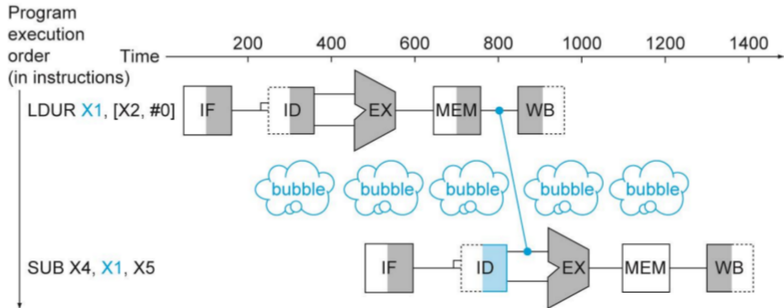
```
int a, b, c, d, e, f;
a = b + e;
c = b + f;
```

Example: We assume the variables are stored in memory, starting from the location held in register `R0`. Here is the naive Assembler code:

```
LDR  R1, [R0, #4]    @ load b
LDR  R2, [R0, #16]   @ load e
ADD  R3, R1, R2      @ b + e
STR  R3, [R0, #0]    @ store a
LDR  R4, [R0, #20]   @ load f
ADD  R5, R1, R4      @ b + f
STR  R5, [R0, #12]   @ store c
```

Can you spot the data hazard in this example?

A Graphical Representation of a Load-Store Hazard



[0] From Patterson & Hennessy, Chapter 4

Example: Reordering Code to Avoid Pipeline Stalls

Example: Translate the following C expression into Assembler:

```
int a, b, c, d, e, f;
a = b + e;
c = b + f;
```

Example: The reordered Assembler code, eliminating the data hazard:

```
LDR  R1, [R0, #4]    @ load b
LDR  R2, [R0, #16]   @ load e
LDR  R4, [R0, #20]   @ load f; moved_up
ADD  R3, R1, R2      @ b + e
STR  R3, [R0, #0]    @ store a
ADD  R5, R1, R4      @ b + f
STR  R5, [R0, #12]   @ store c
```

Moving the third `LDR` instruction upward, makes its result available soon enough to avoid a pipeline stall.

HERIOT WATT UNIVERSITY

Summary: Processor Architecture and Pipelining

- Modern ("super-scalar") processors can execute several instructions at the same time, by organising the execution of an instruction into several stages and using a **pipeline** structure.
- This exploits **instruction-level** parallelism and boosts performance.
- However, there is a risk of control and data hazards, leading to reduced performance, e.g. due to poor branch prediction
- Knowing these risks, you can develop faster code!
- These code transformations are often done internally by the compiler.