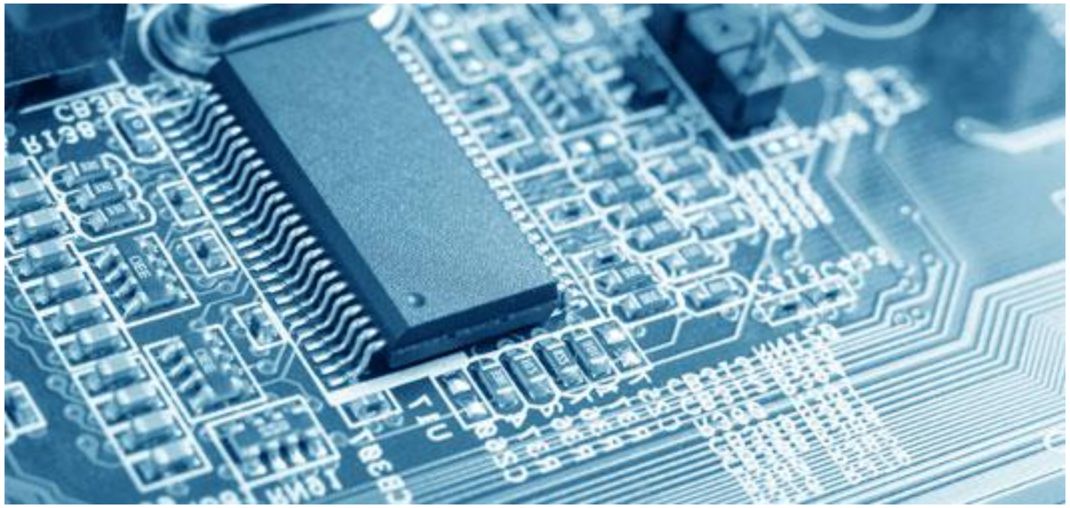
NAZARBAYEV UNIVERSITY
SCHOOL OF SCIENCE AND TECHNOLOGY



CSCI502 – Hardware/Software Co-Design

Lecture 3 – GNU Software Development Toolchain. Hardware for Embedded Systems: Pipelining, Memory-Mapped I/O

21 January 2019

Course Logistics

Reference Reading (available in Moodle and library):
Real-Time Embedded Systems: Chapter 2
Exploring BeagleBone. 2nd edition: Chapter 5
Logic and Computer Design Fundamentals. 5th edition: Chapters 10 – 11 (slide relevant material)

Programming Languages for Embedded Systems based on Beagle Boards

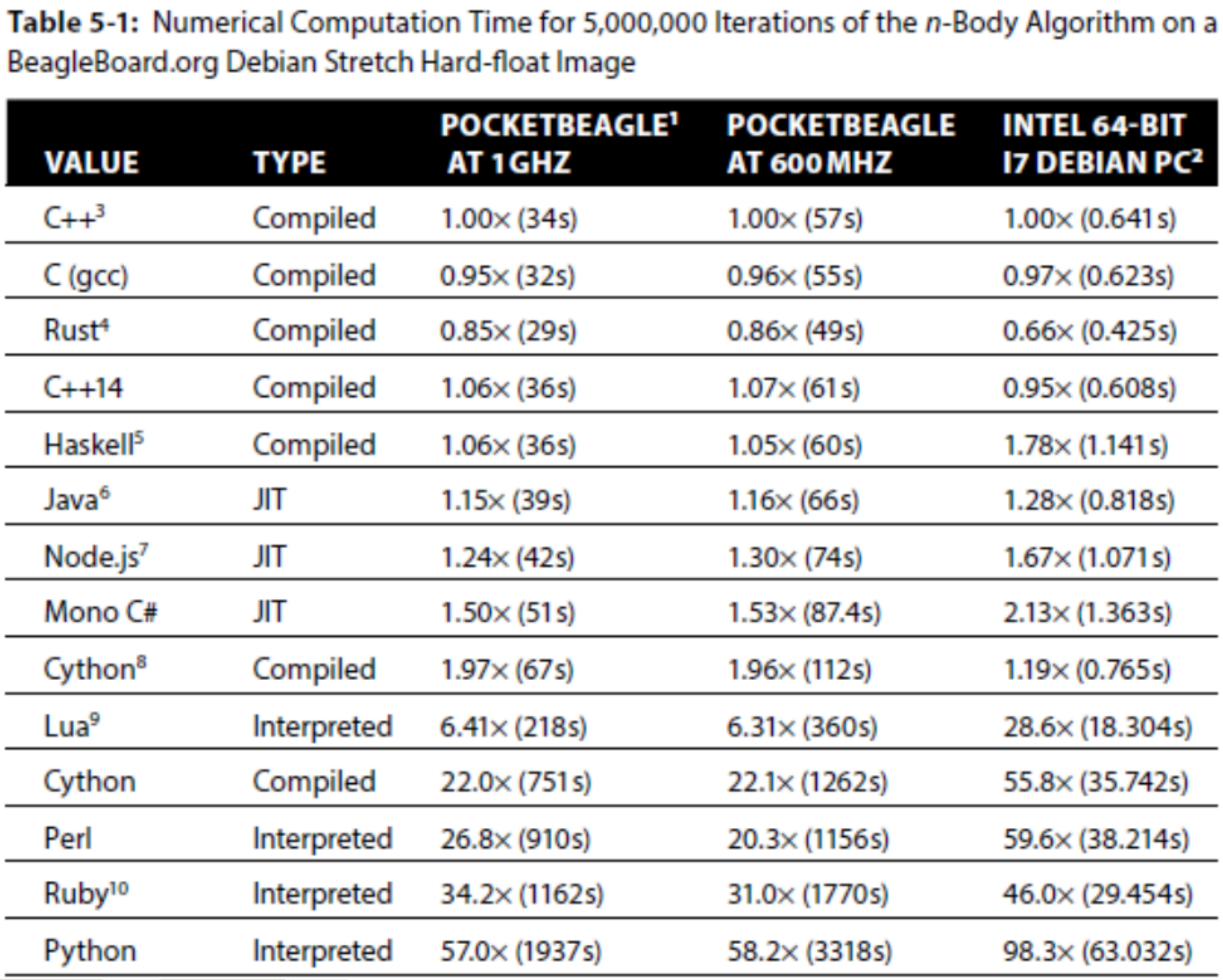
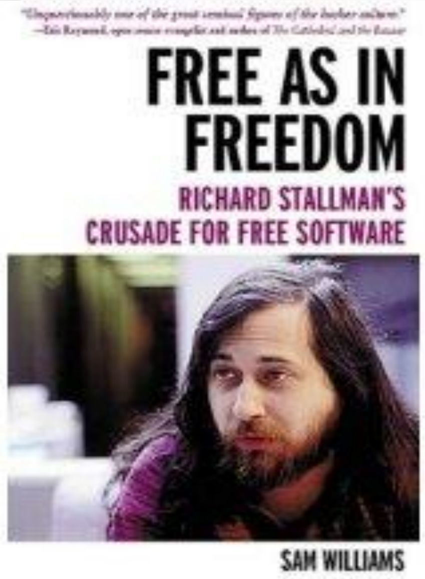


Table 5-1: Numerical Computation Time for 5,000,000 Iterations of the *n*-Body Algorithm on a BeagleBoard.org Debian Stretch Hard-float Image

VALUE	TYPE	POCKETBEAGLE[1] AT 1 GHZ	POCKETBEAGLE AT 600 MHZ	INTEL 64-BIT I7 DEBIAN PC[2]
C++[3]	Compiled	1.00× (34s)	1.00× (57s)	1.00× (0.641 s)
C (gcc)	Compiled	0.95× (32s)	0.96× (55s)	0.97× (0.623s)
Rust[4]	Compiled	0.85× (29s)	0.86× (49s)	0.66× (0.425s)
C++14	Compiled	1.06× (36s)	1.07× (61 s)	0.95× (0.608s)
Haskell[5]	Compiled	1.06× (36s)	1.05× (60s)	1.78× (1.141 s)
Java[6]	JIT	1.15× (39s)	1.16× (66s)	1.28× (0.818s)
Node.js[7]	JIT	1.24× (42s)	1.30× (74s)	1.67× (1.071 s)
Mono C#	JIT	1.50× (51 s)	1.53× (87.4s)	2.13× (1.363s)
Cython[8]	Compiled	1.97× (67s)	1.96× (112s)	1.19× (0.765s)
Lua[9]	Interpreted	6.41× (218s)	6.31× (360s)	28.6× (18.304s)
Cython	Compiled	22.0× (751 s)	22.1× (1262s)	55.8× (35.742s)
Perl	Interpreted	26.8× (910s)	20.3× (1156s)	59.6× (38.214s)
Ruby[10]	Interpreted	34.2× (1162s)	31.0× (1770s)	46.0× (29.454s)
Python	Interpreted	57.0× (1937s)	58.2× (3318s)	98.3× (63.032s)

Linux History: GNU project

- Established in 1984 by Richard Stallman, who believes that software should be free from restrictions against copying or modification in order to make better and efficient computer programs
- Aim to create a free UNIX like OS, consisting of a kernel and all associated software packages



GNU is a recursive acronym for "GNU's Not Unix"

Aim at developing a complete Unix-like operating system which is free for copying and modification

Companies make their money by maintaining and distributing the software, e.g. optimally packaging the software with different tools (Redhat, Slackware, Mandrake, SuSE, etc)

Stallman built the first free GNU C Compiler in 1991. But still, an OS was yet to be developed

C/C++ HelloWorld Examples

Listing 5-7: chp05/overview/helloworld.c

```c
#include <stdio.h>
int main(int argc, char *argv[]){
    printf("Hello World!\n");
    return 0;
}
```

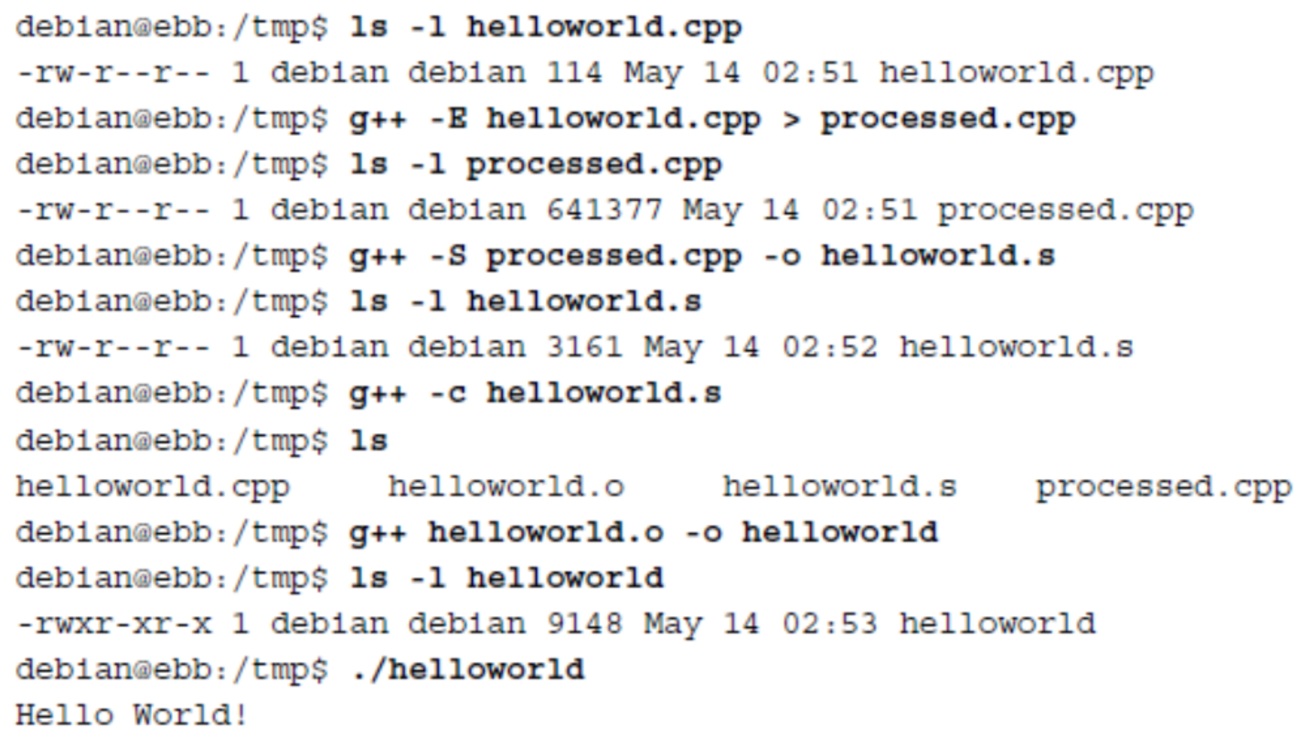
Listing 5-8: chp05/overview/helloworld.cpp

```cpp
#include<iostream>
int main(int argc, char *argv[]){
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

```
.../chp05/overview$ gcc helloworld.c -o helloworldc
.../chp05/overview$ ./helloworldc
Hello World!
.../chp05/overview$ g++ helloworld.cpp -o helloworldcpp
.../chp05/overview$ ./helloworldcpp
Hello World!
```

C/C++ on the Beagle Boards: GNU Software Development Toolchain

- There are a few intermediate steps that are not obvious in the preceding example, as the intermediate stage outputs are not retained by default.

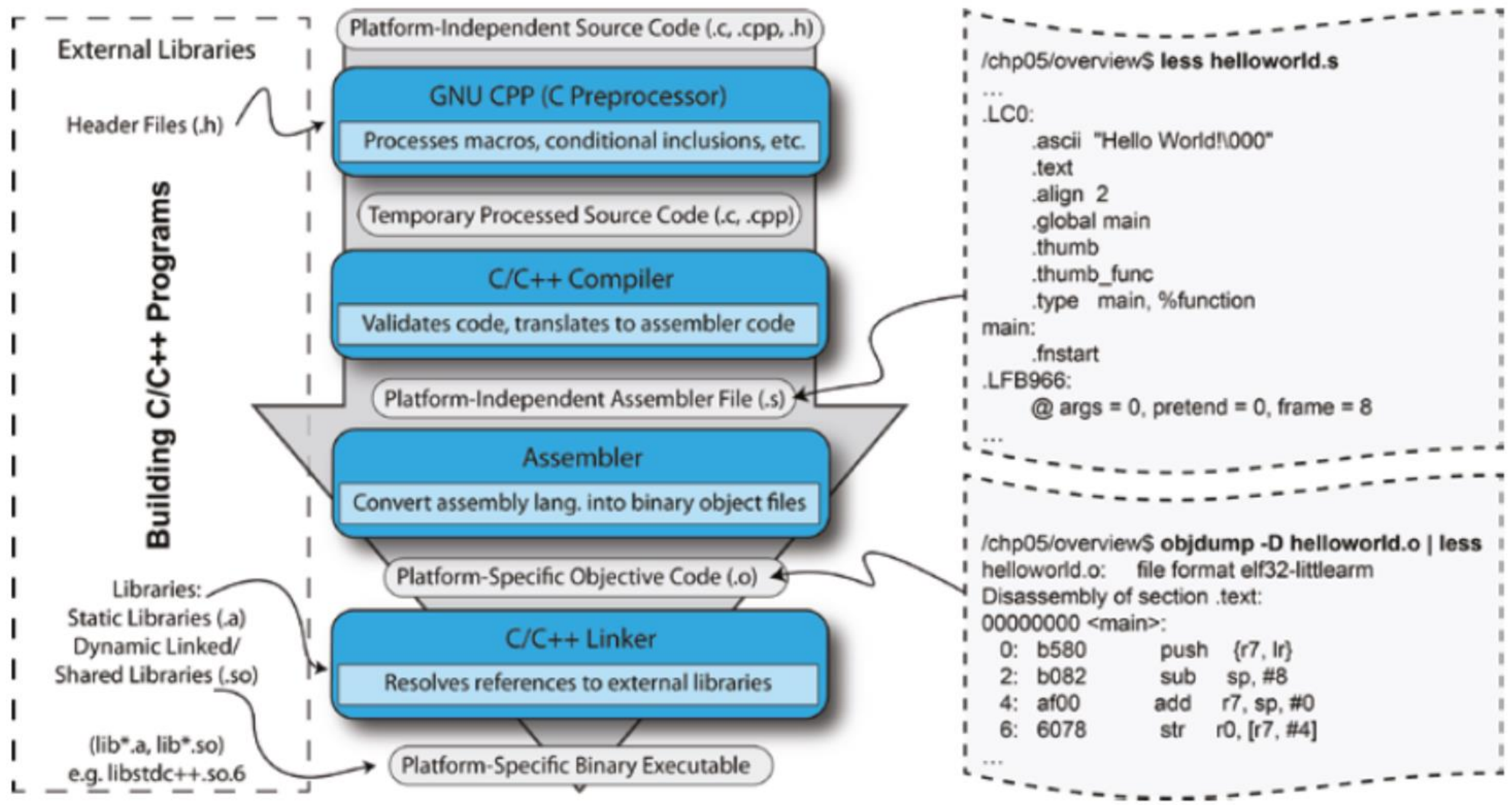


```
debian@ebb:/tmp$ ls -l helloworld.cpp
-rw-r--r-- 1 debian debian 114 May 14 02:51 helloworld.cpp
debian@ebb:/tmp$ g++ -E helloworld.cpp > processed.cpp
debian@ebb:/tmp$ ls -l processed.cpp
-rw-r--r-- 1 debian debian 641377 May 14 02:51 processed.cpp
debian@ebb:/tmp$ g++ -S processed.cpp -o helloworld.s
debian@ebb:/tmp$ ls -l helloworld.s
-rw-r--r-- 1 debian debian 3161 May 14 02:52 helloworld.s
debian@ebb:/tmp$ g++ -c helloworld.s
debian@ebb:/tmp$ ls
helloworld.cpp    helloworld.o    helloworld.s    processed.cpp
debian@ebb:/tmp$ g++ helloworld.o -o helloworld
debian@ebb:/tmp$ ls -l helloworld
-rwxr-xr-x 1 debian debian 9148 May 14 02:53 helloworld
debian@ebb:/tmp$ ./helloworld
Hello World!
```

The text format output after preprocessing can be seen by typing **less processed.cpp**, where you will see the necessary header files pasted in at the top of the code.

C/C++ on the Beagle Boards: GNU Software Development Toolchain

- The full build process from preprocessing right through to linking



External Libraries
Header Files (.h)
Building C/C++ Programs
Libraries:
Static Libraries (.a)
Dynamic Linked/
Shared Libraries (.so)
(lib*.a, lib*.so)
e.g. libstdc++.so.6
Platform-Independent Source Code (.c, .cpp, .h)
GNU CPP (C Preprocessor)
Processes macros, conditional inclusions, etc.
Temporary Processed Source Code (.c, .cpp)
C/C++ Compiler
Validates code, translates to assembler code
Platform-Independent Assembler File (.s)
Assembler
Convert assembly lang. into binary object files
Platform-Specific Objective Code (.o)
C/C++ Linker
Resolves references to external libraries
Platform-Specific Binary Executable

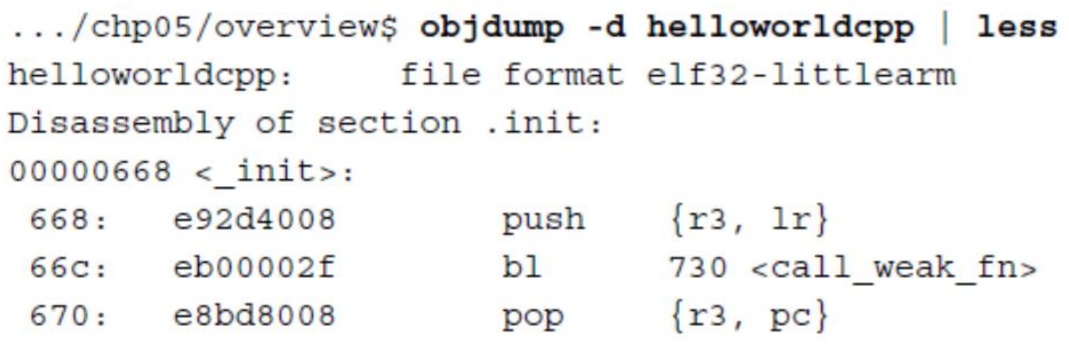
```
/chp05/overview$ less helloworld.s
...
.LC0:
        .ascii  "Hello World!\000"
        .text
        .align  2
        .global main
        .thumb
        .thumb_func
        .type   main, %function
main:
        .fnstart
.LFB966:
        @ args = 0, pretend = 0, frame = 8
...
```

```
/chp05/overview$ objdump -D helloworld.o | less
helloworld.o:     file format elf32-littlearm
Disassembly of section .text:
00000000 <main>:
  0:  b580        push   {r7, lr}
  2:  b082        sub    sp, #8
  4:  af00        add    r7, sp, #0
  6:  6078        str    r0, [r7, #4]
...
```

Figure 5-1: Building C/C++ applications on the Beagle boards

C/C++ on the Beagle Boards: GNU Software Development Toolchain

- At first, a platform-independent assembler code is generated (e.g., the **helloworld.s** file)
- This `.s` file is then passed to the assembler, which converts the platform independent instructions into binary instructions (object file) for the Beagle board (e.g., the **helloword**`.o` file).
- After linking the final executable code, `helloworld` contains the target-specific assembly language code that has been combined with the libraries, statically and dynamically as r



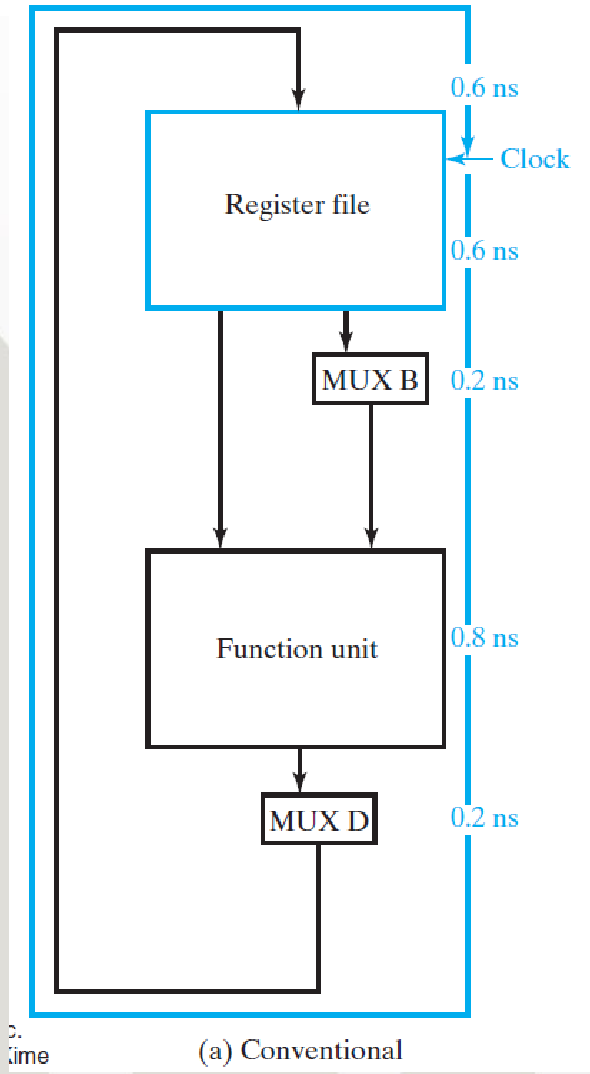
```
.../chp05/overview$ objdump -d helloworldcpp | less
helloworldcpp:     file format elf32-littlearm
Disassembly of section .init:
00000668 <_init>:
 668:    e92d4008         push    {r3, lr}
 66c:    eb00002f         bl      730 <call_weak_fn>
 670:    e8bd8008         pop     {r3, pc}
```

- The first column is the memory address, which steps by 4 bytes (32-bits) between each instruction (i.e., `66c − 668 = 4`).
- The second column is the full 4-byte instruction at that address.
- The third and fourth columns are the human-readable version of the second column

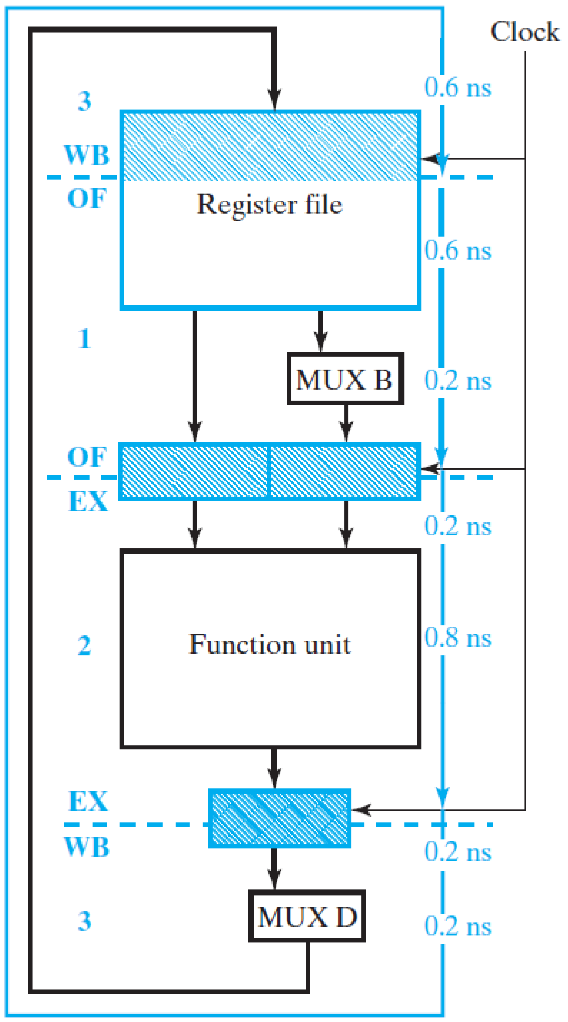
Hardware Components

- Embedded systems interact closely with input and output devices including sensors and actuators
- Hardware structure can affect timing and require special software design techniques

Datapath Timing



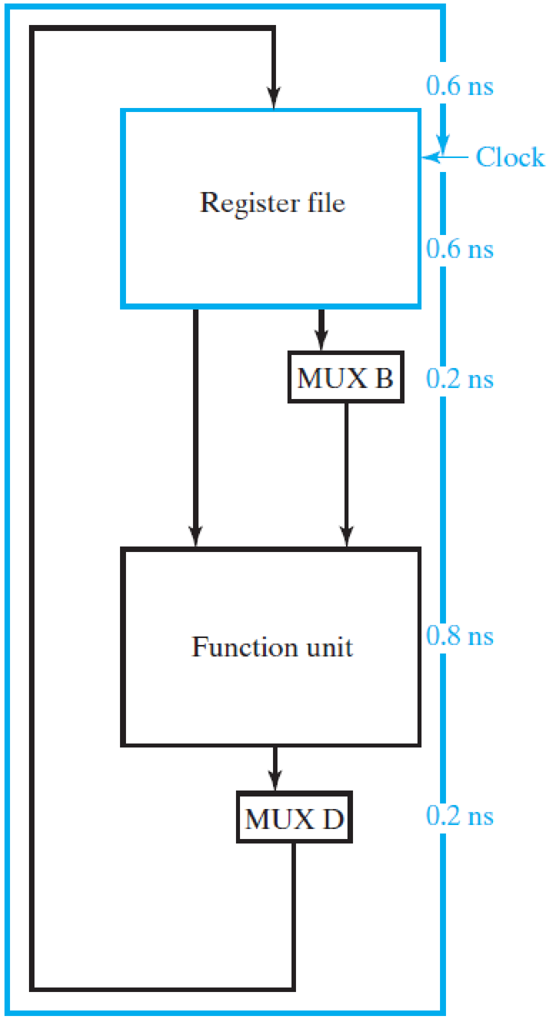
(a) Conventional



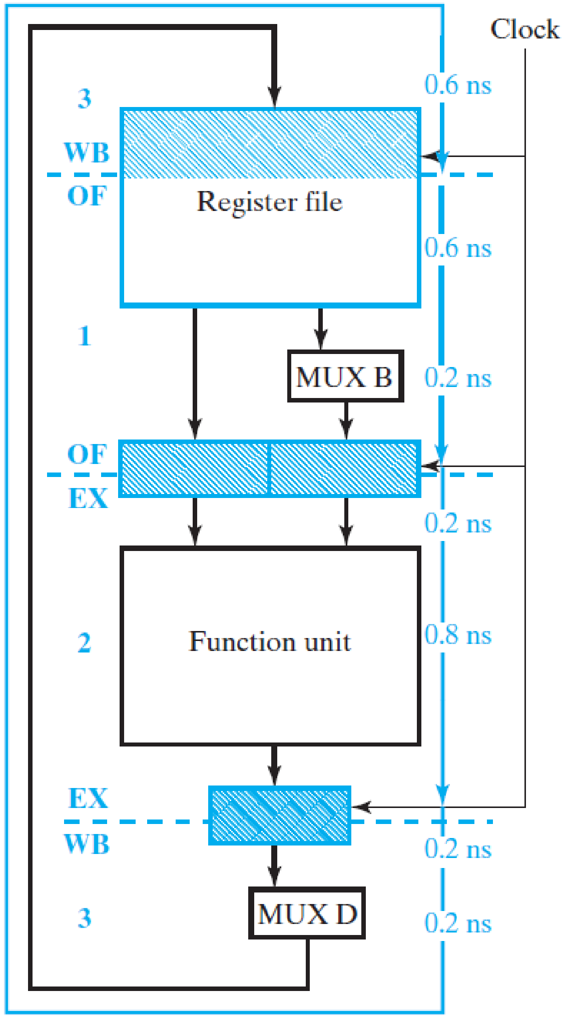
(b) Pipelined

- ❑ Fig. A (left) illustrates maximum delay values for each of the components of a typical datapath
- ❑ About 2.4 ns is needed to perform a single microoperation
- ❑ The max rate of microoperation execution is the inverse of 2.4 ns = 416.7MHz - max. frequency at which the clock can be operated with clock period 2.8 ns
- ❑ To reduce clock period can be done by breaking up the 2.8 ns delay with registers – **pipelined datapath**

Datapath Timing



(a) Conventional



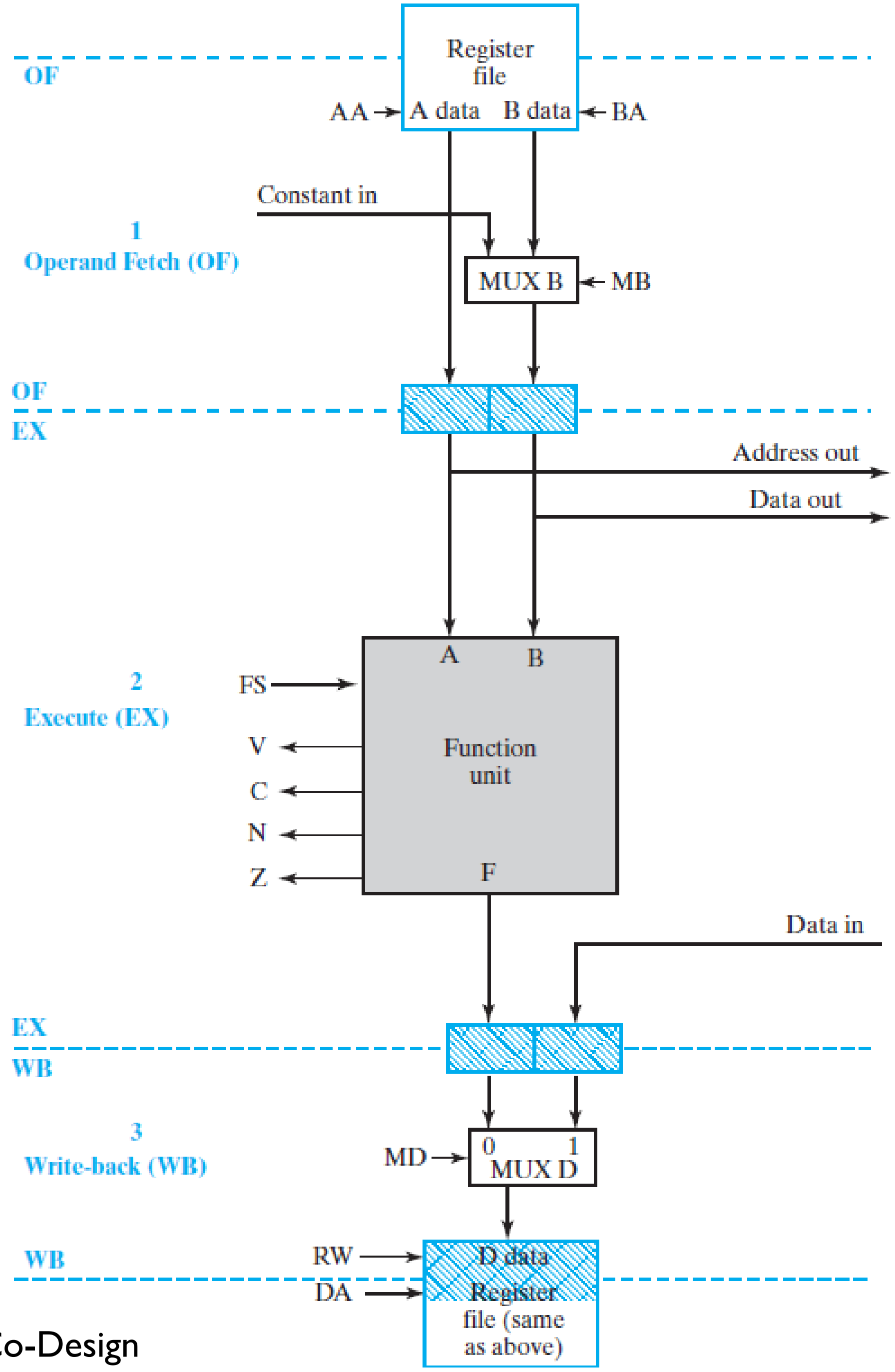
(b) Pipelined

- Fig. B shows desired structure with division to thee stages:
1. Operation fetch (OF)
2. Execution (EX)
3. Write-back (WB)
- OF has delay for reading the register file and selecting MUX B = 0.8 ns
- EX has 1.0 ns delay mainly due Function unit
- WB has 1.0 ns in total for selecting MUX D and writing back to the register file
- Thus, all delays are at most 1.0 ns allowing clock period of 1.0 ns and max. clock frequency 1 GHz

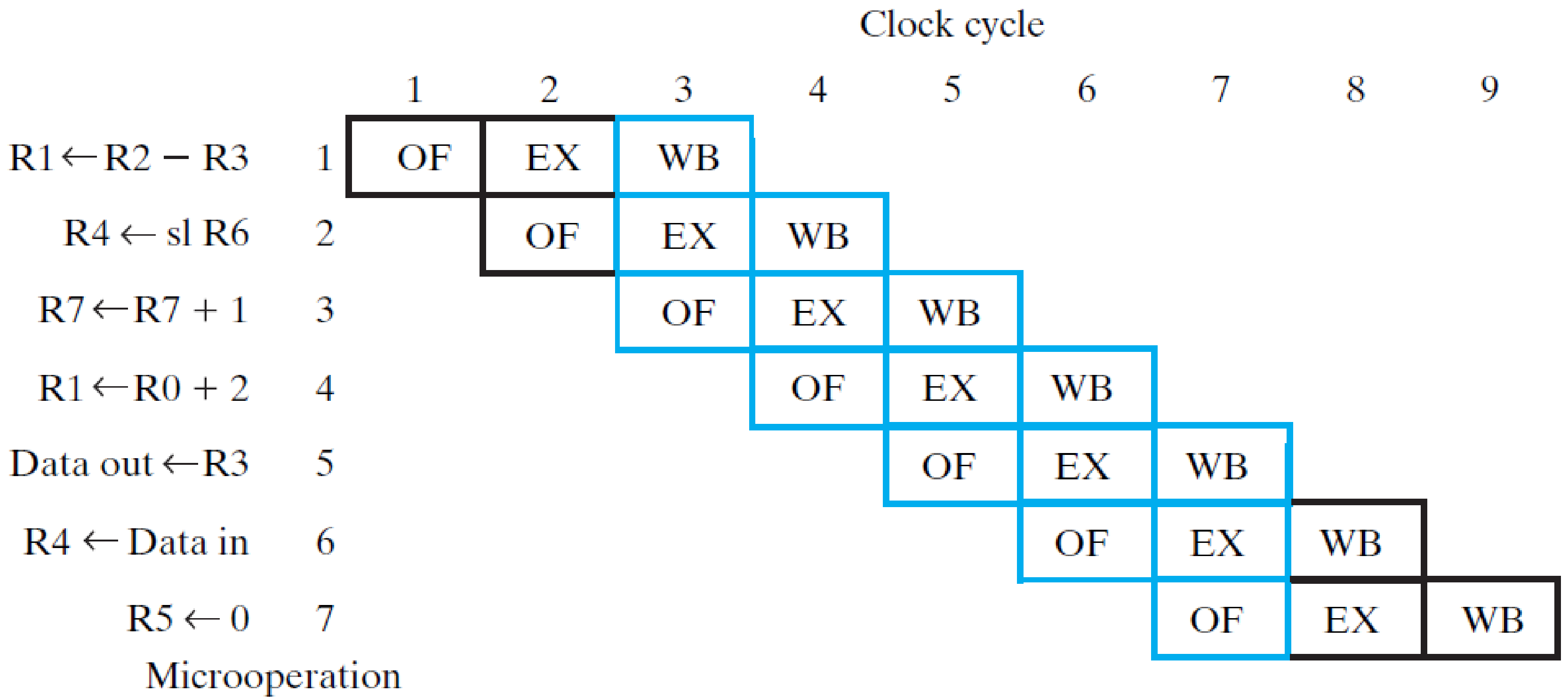
c.
Kime

Pipelined Datapath

- More detailed diagram of the pipelined datapath
- The register file is shown twice
- Pipelining imparts an implicit execution parallelism in the different cycles of processing an instruction.
- With pipelining, more instructions can be processed in different cycles simultaneously, improving processor performance.



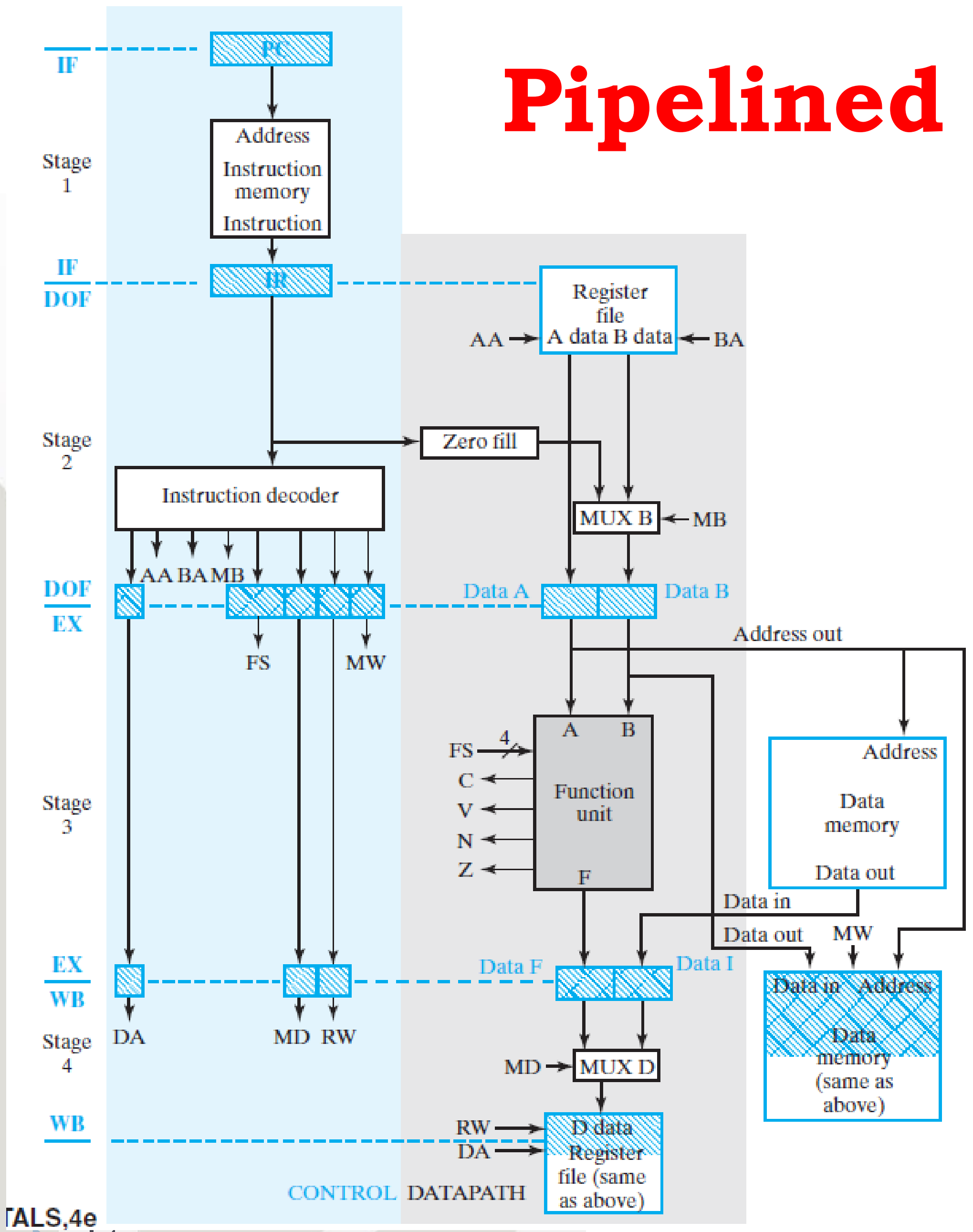
Pipeline Execution



Microoperation		Clock cycle 1	2	3	4	5	6	7	8	9
R1←R2 − R3	1	OF	EX	WB						
R4 ← sl R6	2		OF	EX	WB					
R7←R7 + 1	3			OF	EX	WB				
R1←R0 + 2	4				OF	EX	WB			
Data out ←R3	5					OF	EX	WB		
R4 ← Data in	6						OF	EX	WB	
R5 ← 0	7							OF	EX	WB

- 7 microoperations requires 9 clock cycles to execute completely.
- Time required = 9 x 1ns = 9 ns compared to 7 x 2.4 ns = 16.8 ns for the traditional datapath (1.9 times faster using pipeline)

Pipelined Control



- The control has added Stage 1 - Instruction Fetch (IF)
- Stage 2 – Decode and Operand Fetch (DOF)
- Location of the pipeline platforms balance partitioning of the delays with maximum no more than1 ns.
- Max clock frequency is 1 GHz
- An instruction takes 4 x 1 ns = 4 ns to execute

Pipelining

- A new instruction is completed at the rate of one clock cycle

- Pipelining can actually degrade performance – if any of the instructions in the pipeline are a branch instruction, the prefetched instructions further in the pipeline are no longer valid and must be flushed.

RISC Architecture

- Reduced instruction set computers (RISC):

1. Simple instruction taking one clock cycle
2. Memory access by load/store instructions only
3. Highly pipelining instruction processing
4. Hard-wired control unit
5. Small number of instructions
6. Instructions are fixed format and length
7. Few addressing modes
8. Multiple sets of work registers
9. Complexity handled by compilers and software

Example:
```
LOAD R1,A     //load A into register R1
LOAD R2, B    //load B into register R2
PROD R1, R2   //multiply A and B, result saved in R1
STOR R1,A     //store A*B into a memory location
```

CISC Architercture

- Complex instruction set computers (CISC):

1. Complex instructions take multiple clock cycles
2. Practically any instruction can reference memory
3. Microprogrammed control unit
4. Large number of instructions
5. Instructions are of variable format and length
6. Great variety of addressing modes
7. Single set of work registers
8. Complexity handled by the microprogram and hardware

Example: MULT A, B – complex instruction.

Operates directly on memory and does not require explicit call of loading and storing functions.

CISC vs RISC

- RISC has fewer instructions, complex operations are sequence of simple instructions
- RISC's major advantage in real-time systems:
  - is the shorter average instruction execution time than for CISCs
  - shorter interrupt latency, hence shorter response times
- RISC's downsides:
  - Processors associated with cashes and multistage pipelines
  - Response time increases with low cache hit ratios
- Often tolerable in firm and soft RTOS

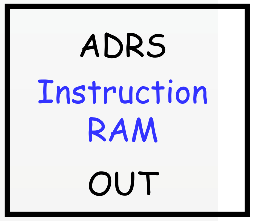
Memory Technologies: ROM

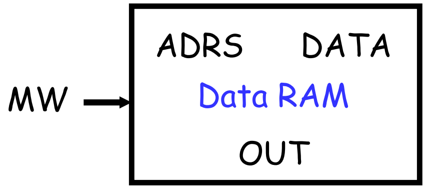
Two classes of memory :
RAM – random access memory
ROM – read-only memory

- Electrically erasable programmable ROM (EEPROM) and Flash (ROM) can be rewritten similar to RAM devices
- Erasing and writing process is much slower comparing to RAM and limited to 100000 -1000000 times
- EEPROM is mainly used for nonvolatile program and parameter storing
- Flash – application program and data records

Where Does the Program Go?

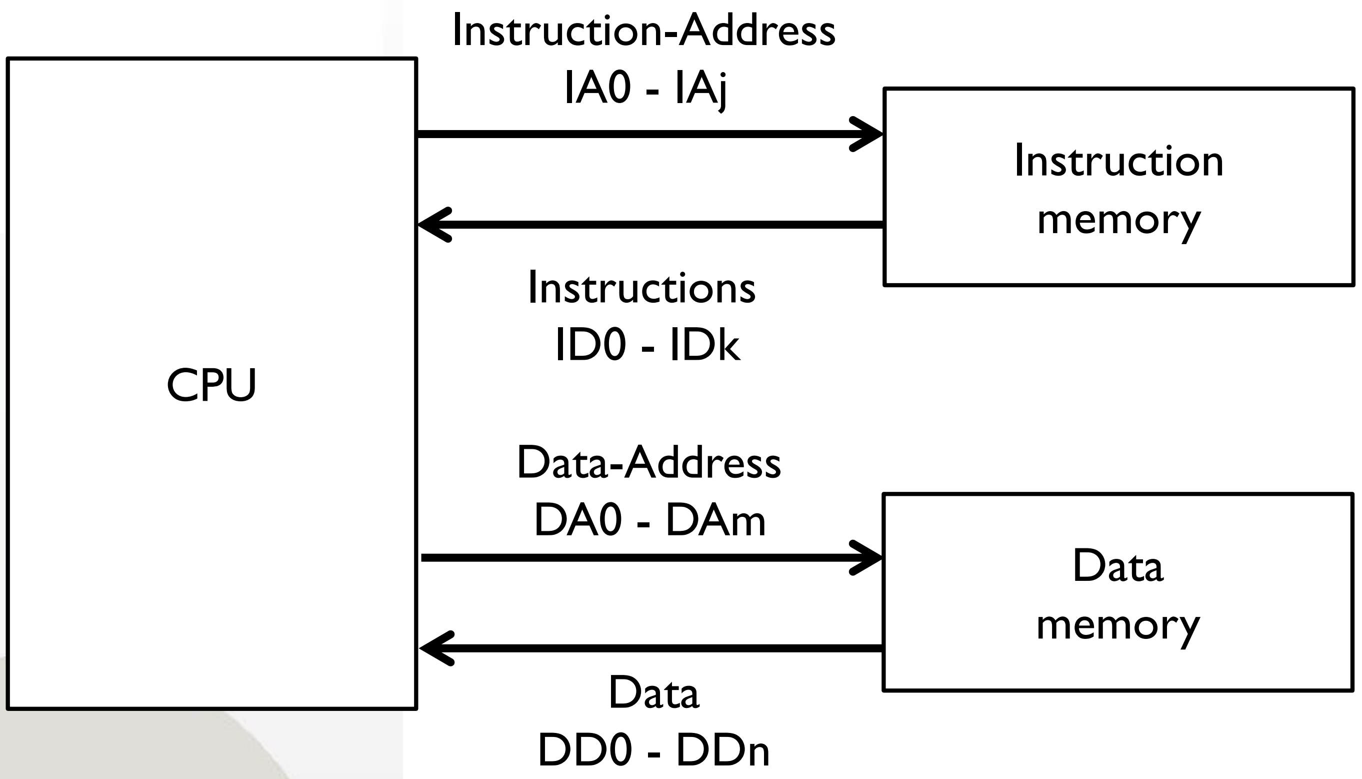
- Harvard architecture includes two memory units.
  - An instruction memory holds the program.
  - A separate data memory is used for computations.
  - **The advantage is that we can read an instruction *and* load or store data in the same clock cycle**.
- For simplicity, our diagrams do not show any WR or DATA inputs to the instruction memory.





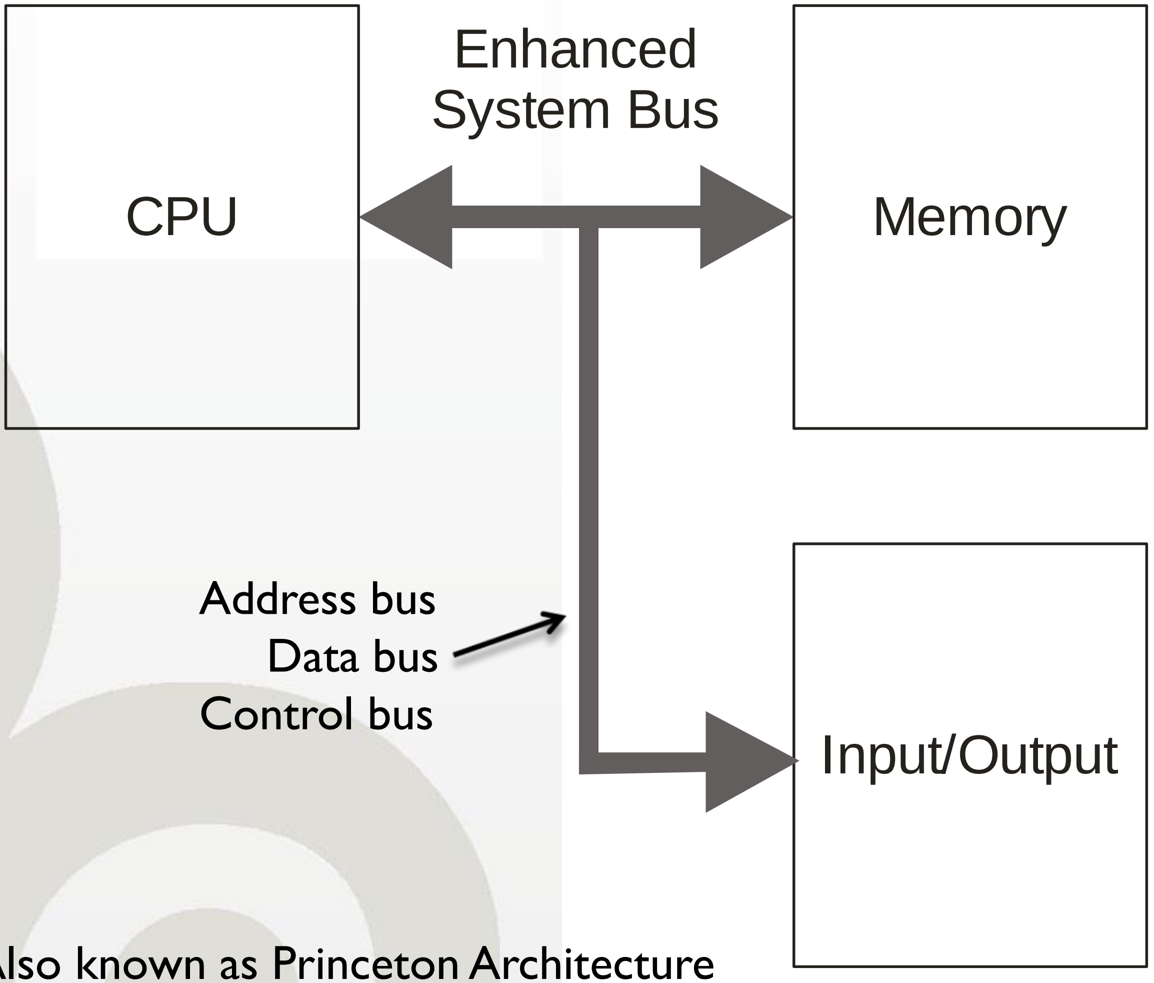
- Caches in modern CPUs often feature a Harvard architecture like this.
- However, there is usually a single main memory that holds both program instructions and data, in a Von Neumann architecture.

Harvard Architecture



Possible to have different bus widths for instruction and data transfer

Von Neumann Architecture



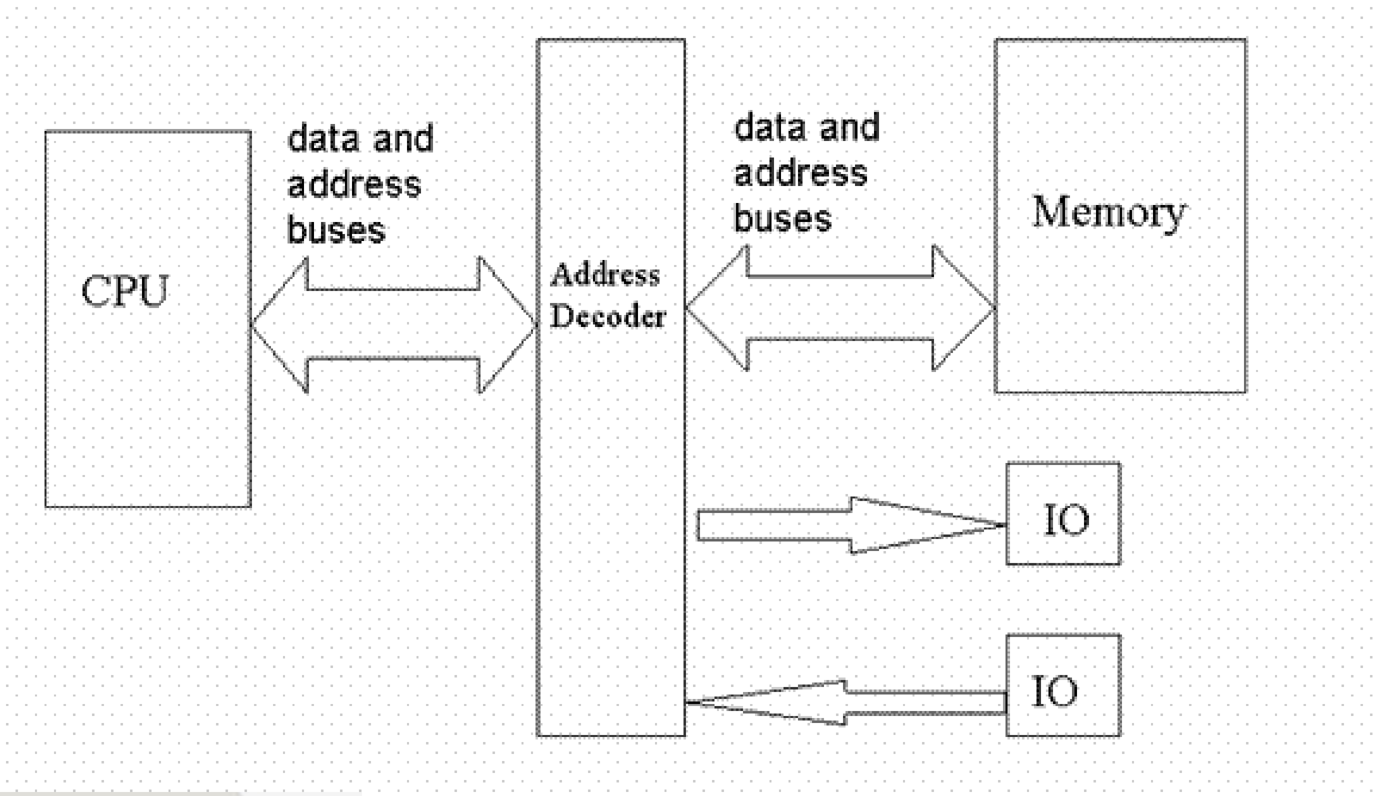
John von Neumann
1903 – 1957

Also known as Princeton Architecture

Memory-Mapped I/O

- In addition to communicating with I/O devices, CPU must communicate with memory unit through an address and data bus
- Memory-mapped I/O provides a data transfer mechanism that is convenient because it does not require the use of special CPU I/O instructions.
- In memory-mapped I/O certain designated locations of memory appear as virtual input/output ports.
- Could be advantageous when implementing efficient device drivers

Memory-Mapped I/O



Input from an appropriate memory-mapped location involves executing a LOAD instruction on a pseudomemory location connected to an input device. Output uses a STORE instruction.

Any Questions?