



NTNU
Norwegian University of
Science and Technology

Lecture 10: Review lecture

Asbjørn Djupdal
ARM Norway, IDI NTNU
2013

Lecture overview

- Exam information
- Curriculum overview
- Other courses

Exam information

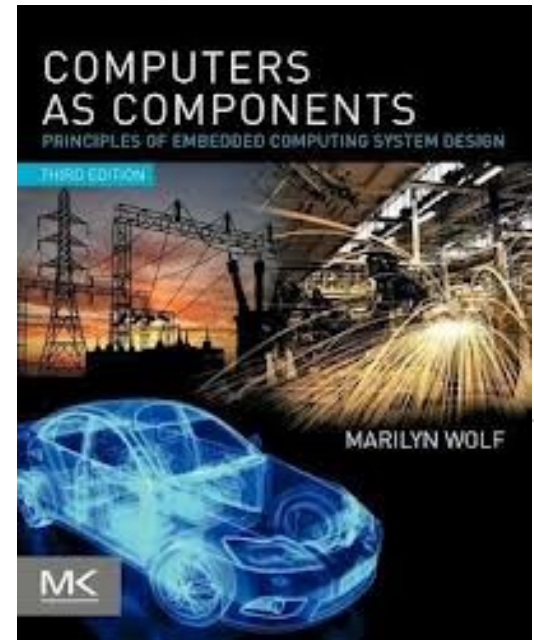
- May 29, 09:00
- Length: 3 hours
- Allowed aids: Category D
 - No written or handwritten support material
 - A specified, simple calculator is permitted
- Type: Written
 - Multiple choice (~25%)
 - Theoretical and practical problems

Grading

- Exercises: 50%
 - Exercise 1: 10%
 - Exercise 2: 20%
 - Exercise 3: 20%
- Exam: 50%

Curriculum

- The entire book
 - Computers as components, Wolf
 - 2nd or 3rd edition
- All slides
 - Including from guest lecturers
- Exercises
 - Documentation and knowledge required to do the exercises
 - Includes understanding C and assembly code



Exam preparation

- Read the curriculum
- Earlier exams and tests are on the webpage

Questions regarding curriculum and exam

- Write your question in an email:
djupdal@idi.ntnu.no
- The reply will go out to all students through It's learning

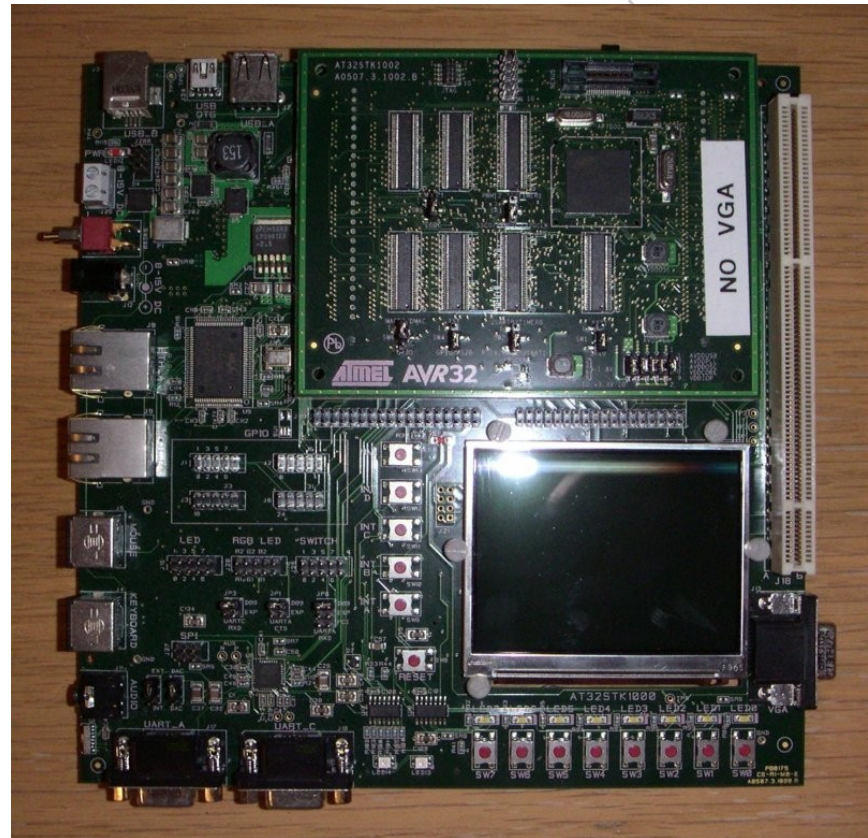
Developing for embedded systems

Exercises

Chapters 1, 4 and 7

Embedded design

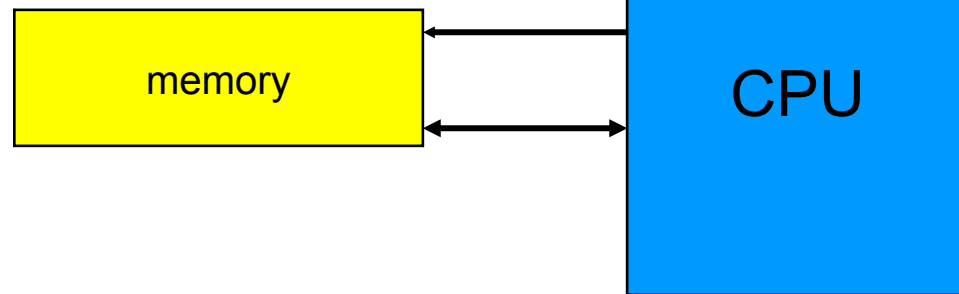
- Dev boards
- Debugging techniques
 - SW: breakpoints
 - HW: ICE, logic analyzer
- Design process



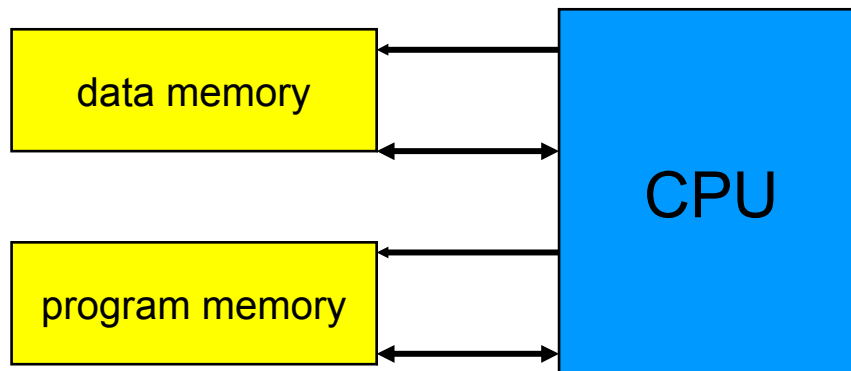
Processors and instructions

Chapters 2, 3

CPU Architecture



von Neuman



Harvard

Instructions

ADC \$01f0

ADD R2, R3

ADD R3, R2, R3

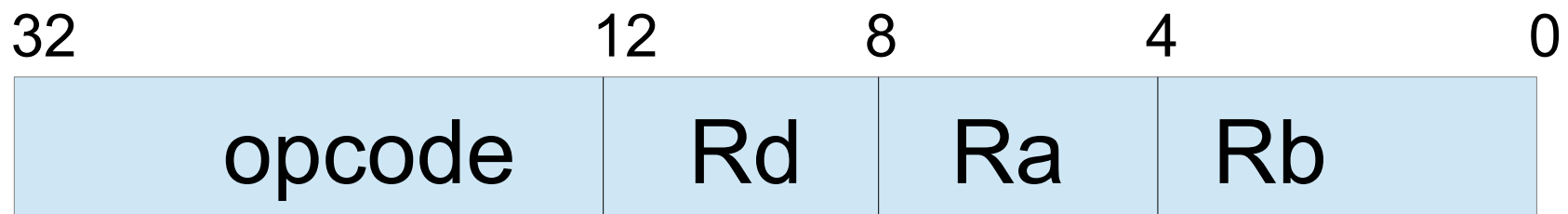
- Number of operands
- Types of operands
- Addressing modes
 - How to store data to memory

RISC Instruction word

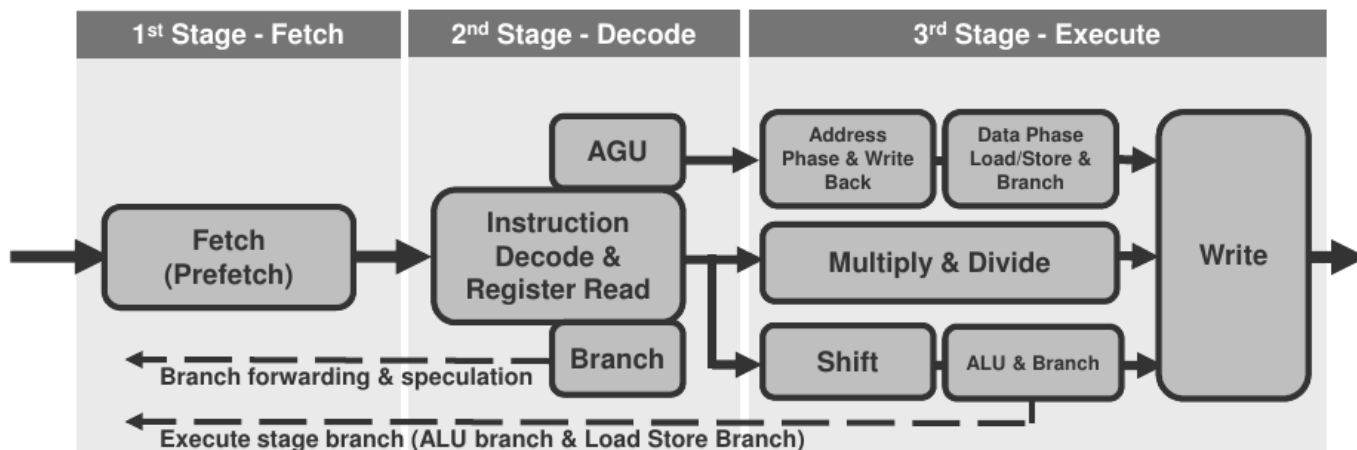
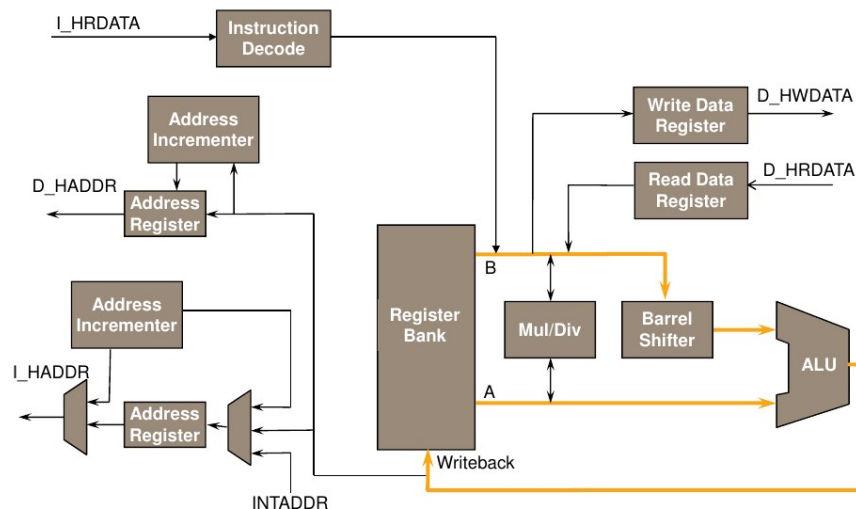
- Fixed length
- A few different formats

ADD R3, R2, R3

e0823003

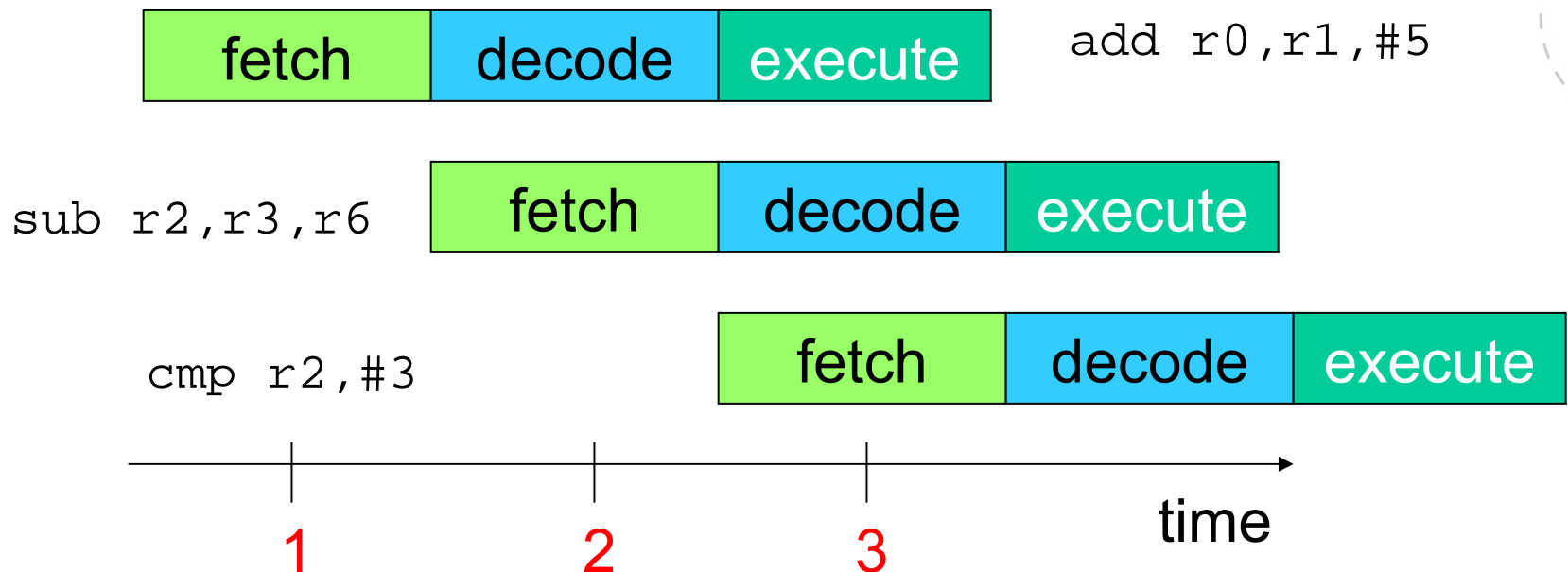


ARM processor organization



Pipelining

- Several instructions are executed at the same time, but at different phases of execution
- Hazards, flushing, bubbles, stalling

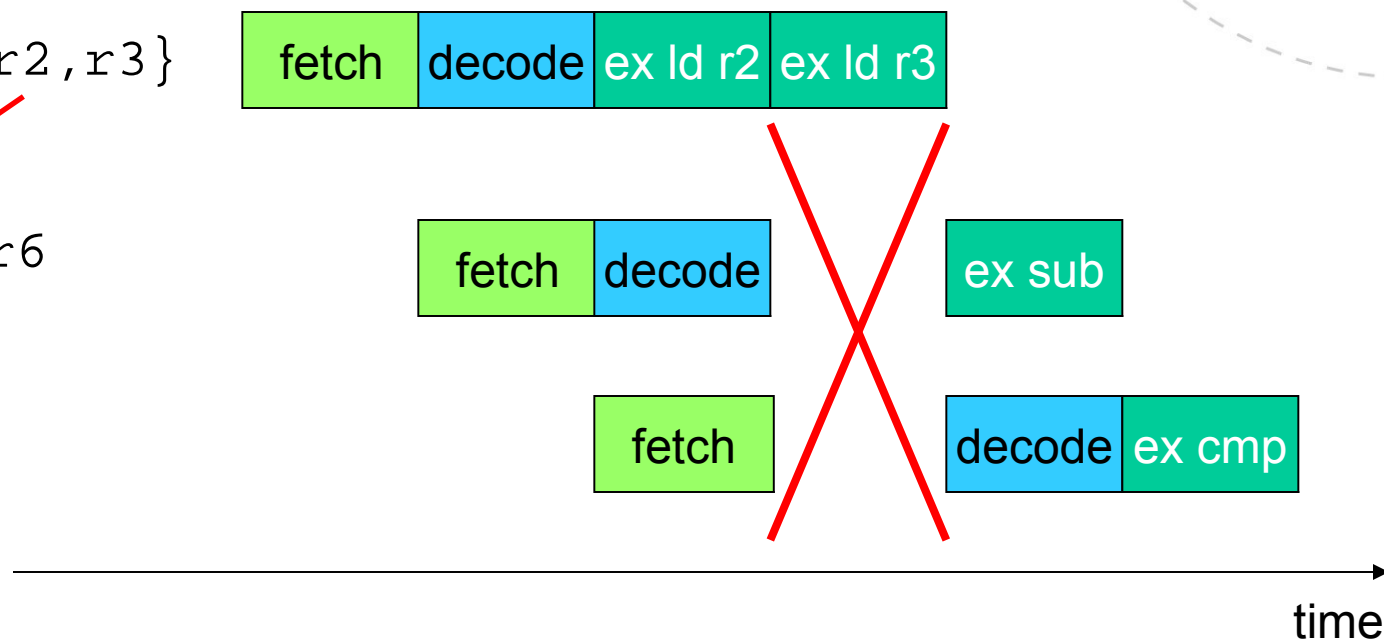


Data hazard

ldmia r0,{r2,r3}

sub r2,r3,r6

cmp r2,#3

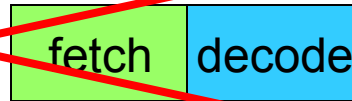


Control hazard

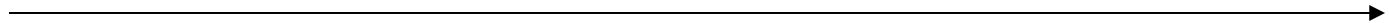
bne foo



~~sub r2,r3,r6~~



add r0,r1,r2



bne foo

sub r2, r3, r6

...

foo:

add r0, r1, r2

Assembly programming

Exercises Chapter 2

Example: C to ARM assembly

```
int a = 1;
int b = 2;
int x;

if(a < b) {
    x = 1;
} else {
    x = 2;
}
```

```
ldr    r1, vars
ldr    r2, vars+4
cmp    r1, r2
movge  r2, #2
movlt  r2, #1
str    r2, vars+8
...
```

vars:

```
.word 1      ; a
.word 2      ; b
.word 0      ; x
```

Subroutines

- Branch-and-link instruction

`BL label`

- Copy PC (R15) to LR (R14)
- Jump to `label`

- Function arguments

- Registers or stack

- Local variables

- Allocate on stack

- Avoid destroying register values

- Push to the stack

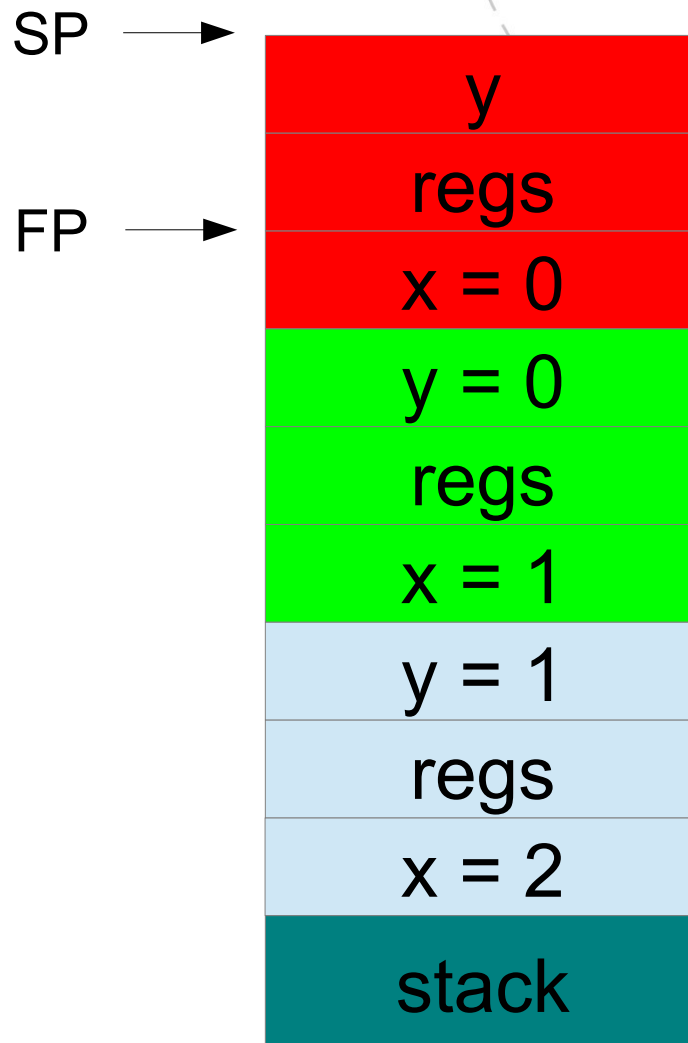
- Return from subroutine

`MOV PC, LR`

Stack frames

```
void test(int x) {  
    int y = x - 1;  
    if(x > 0) {  
        test(y);  
    }  
    return;  
}
```

```
test(2);
```



ARM subroutine example

```
void test(int x) {  
    int y = x - 1;  
    if(x > 0) {  
        test(y);  
    }  
    return;  
}
```

```
void main(void) {  
    test(2);  
}
```

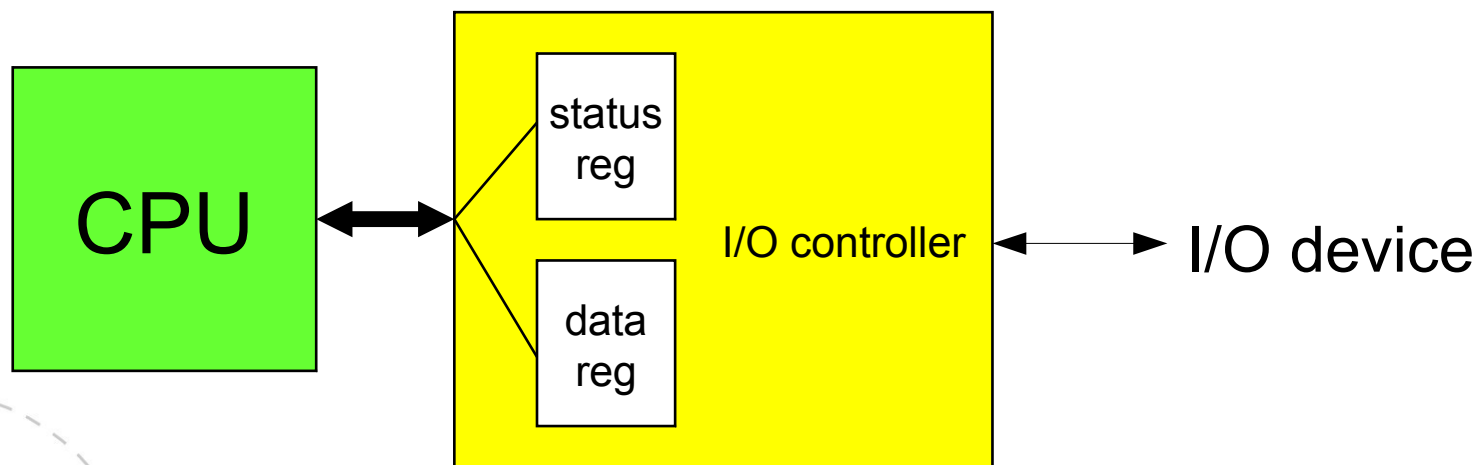
```
test:  
    stmfd sp!, {fp, lr}  
    sub fp, sp, #4  
    sub sp, sp, #4  
  
    sub r1, r0, #1  
    str r1, [fp]  
  
    cmp r0, #0  
    ble end  
  
    ldr r0, [fp]  
    bl test  
  
end:  
    add sp, fp, #4  
    ldmfd sp!, {fp, pc}  
  
main:  
    stmfd sp!, {fp, lr}  
    mov r0, #2  
    bl test  
    ldmfd sp!, {fp, sp}
```

IO controllers and exceptions

Exercises Chapter 3

I/O controllers and devices

- **Memory mapped**
 - General purpose load/store instructions
 - Most common I/O interface
- **Separate I/O address space**
 - Special purpose I/O instructions
 - Intel x86 provides in, out instructions



Polling

```
#define OUT_CHAR (void*)0x1000
#define OUT_STATUS (void*)0x1004

#define READY 0

uint8_t peek(volatile uint8_t *addr) {
    return *addr;
}

void poke(volatile uint8_t *addr, uint8_t val) {
    *addr = val;
}

char *current = string;
while(*current != '\0') {
    while(peek(OUT_STATUS) != READY); // wait until ready
    poke(OUT_CHAR, *current);
    current++;
}
```

Exceptions

- An event that causes CPU to make a jump from its normal execution path
- Types:
 - Hardware exceptions (external)
 - Reset
 - Interrupts
 - Page faults
 - Software exceptions (internal) (traps)
 - Undefined instructions
 - Special instructions for creating exceptions
















Exception behaviour






- Similar to subroutine call mechanism
- Exception forces CPU to:
 - save PC
 - update status register
 - jump to predetermined location (the exception handler)
- After the exception handler is done, the saved PC can be reloaded, and execution continues where it originally was

ARM processor modes

- Normal modes
 - User
 - All user code, not privileged
 - System
- Exception modes
 - Supervisor
 - System call
 - Abort
 - Page miss
 - Undefined
 - Undefined instruction
 - Interrupt
 - Normal interrupt
 - Fast interrupt
 - Interrupts for devices which needs immediate attention

ARM registers

Modes						
<div> <div>Privileged modes</div> <div>Exception modes</div> </div>						
User	System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	 R8_fiq
R9	R9	R9	R9	R9	R9	 R9_fiq
R10	R10	R10	R10	R10	R10	 R10_fiq
R11	R11	R11	R11	R11	R11	 R11_fiq
R12	R12	R12	R12	R12	R12	 R12_fiq
R13	R13	 R13_svc	 R13_abt	 R13_und	 R13_irq	 R13_fiq
R14	R14	 R14_svc	 R14_abt	 R14_und	 R14_irq	 R14_fiq
PC	PC	PC	PC	PC	PC	PC

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		 SPSR_svc	 SPSR_abt	 SPSR_und	 SPSR_irq	 SPSR_fiq

 indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode

ARM exception handling

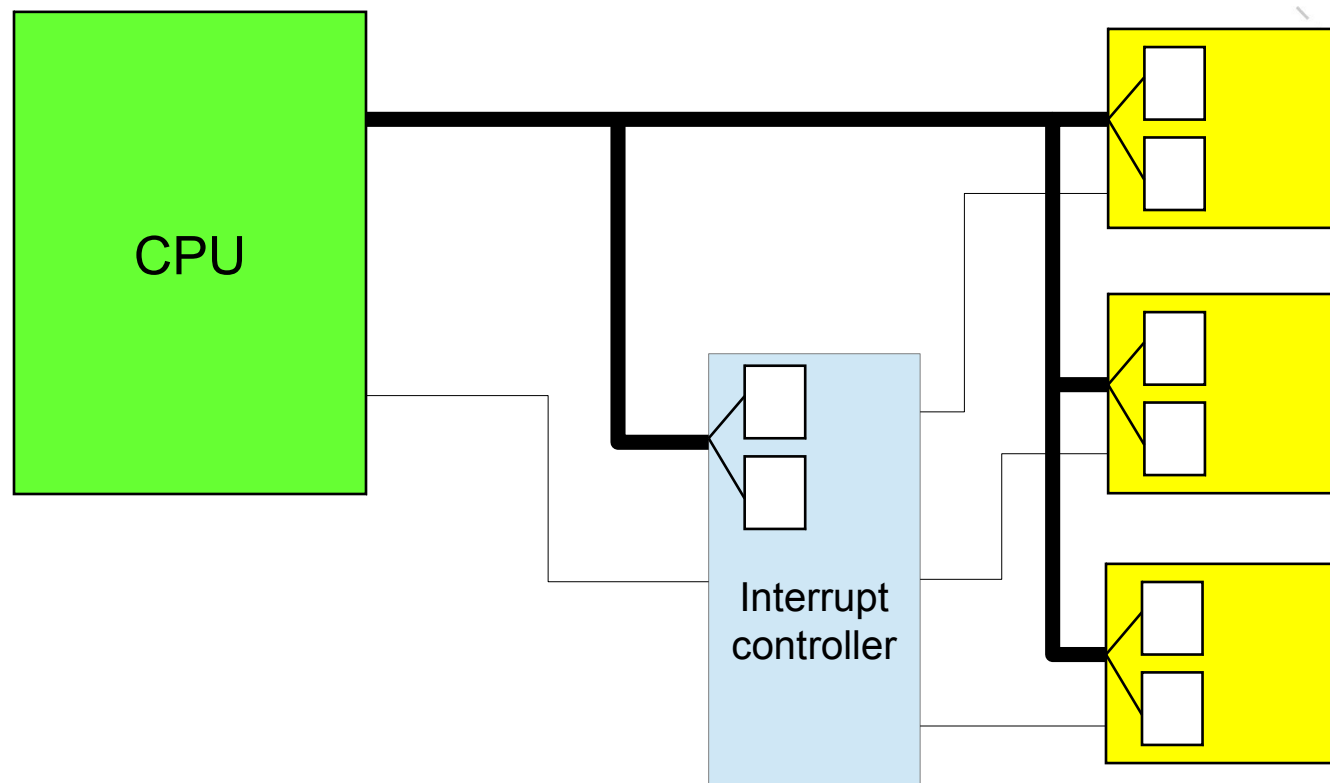
- Vector table at address 0

Vector_Table:

```
LDR pc, Reset_Addr
LDR pc, Undefined_Addr
LDR pc, SVC_Addr
LDR pc, Prefetch_Addr
LDR pc, Abort_Addr
NOP                               ;Reserved vector
LDR pc, IRQ_Addr
LDR pc, FIQ_Addr
```

```
Reset_Addr:    .word Reset_Handler
Undefined_Addr: .word Undefined_Handler
SVC_Addr:      .word SVC_Handler
Prefetch_Addr: .word Prefetch_Handler
Abort_Addr:    .word Abort_Handler
IRQ_Addr:      .word IRQ_Handler
FIQ_Addr:      .word FIQ_Handler
```

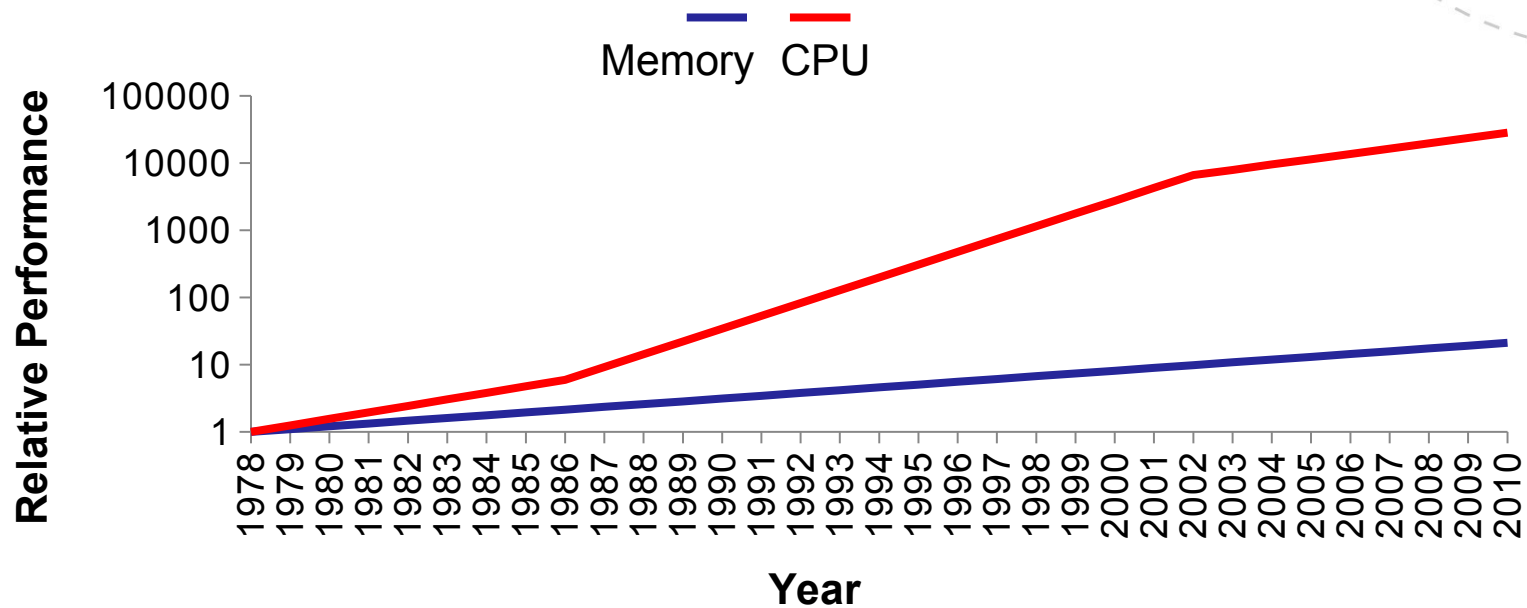
Interrupt controllers



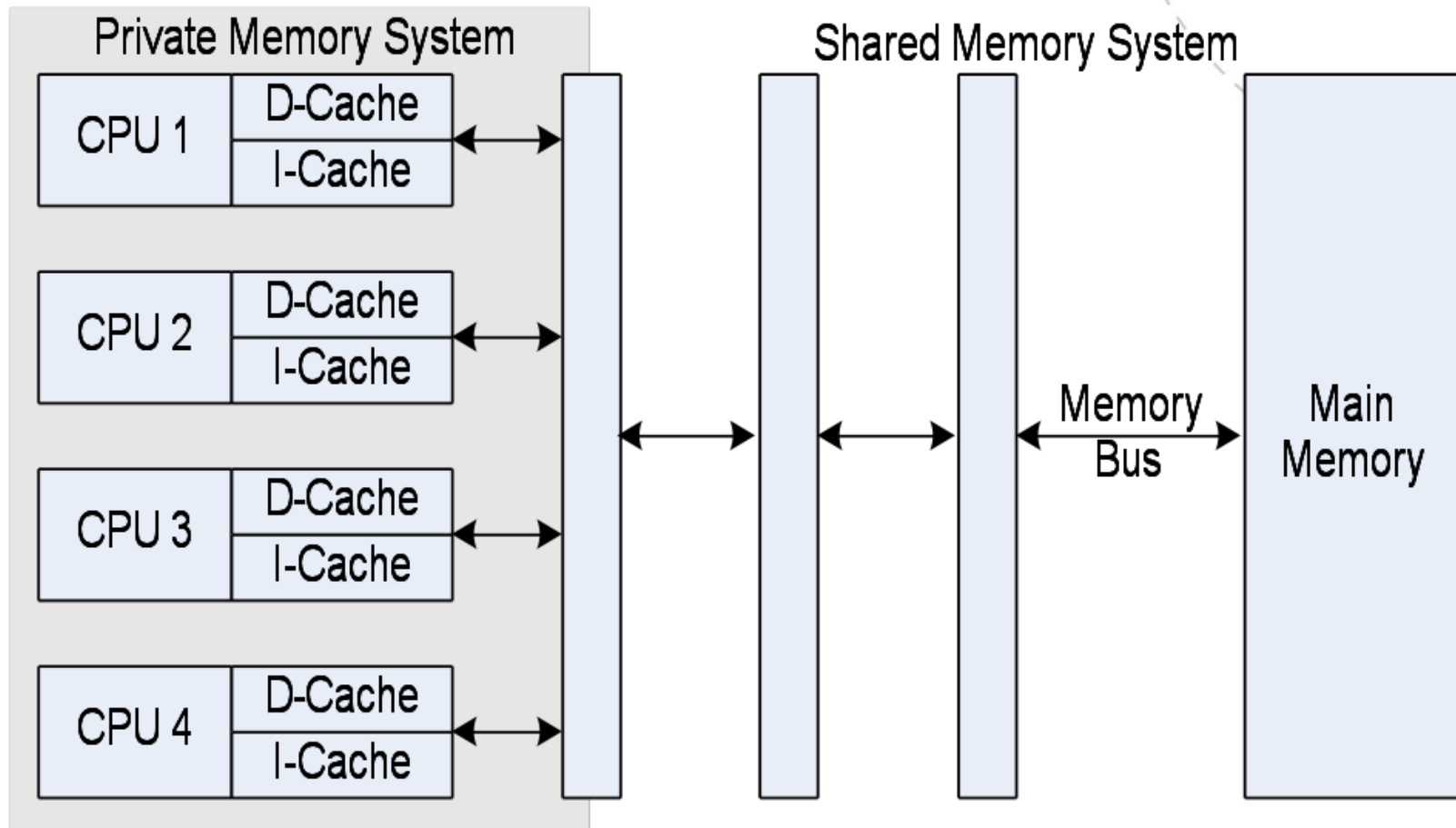
Memory systems and busses

Chapters 3, 4, 8

Why caching?



Multiple layers



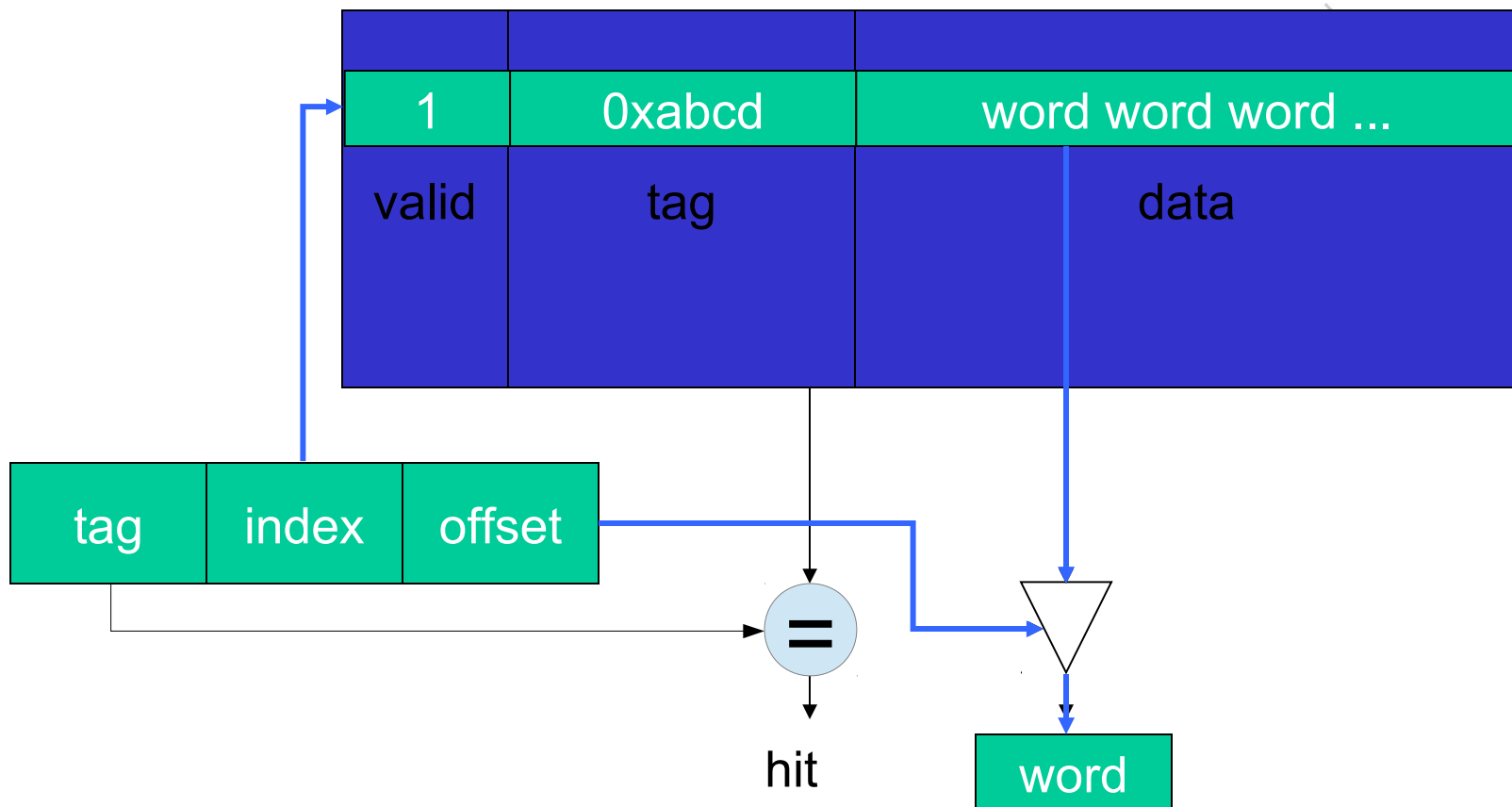
Why caches work

- Principle of locality
 - Temporal locality
 - Spatial locality
- Motivation: Repeatedly processing data in an array
 - C arrays are in contiguous memory
 - Spatial locality: We access memory A, A+1, A+2,...
 - Temporal locality: We access the array often when we have started processing on it

Memory

09	
10	A
11	A+1
12	A+2
13	A+3
14	A+4
15	A+5
16	
17	x
18	y
19	

Direct mapped cache



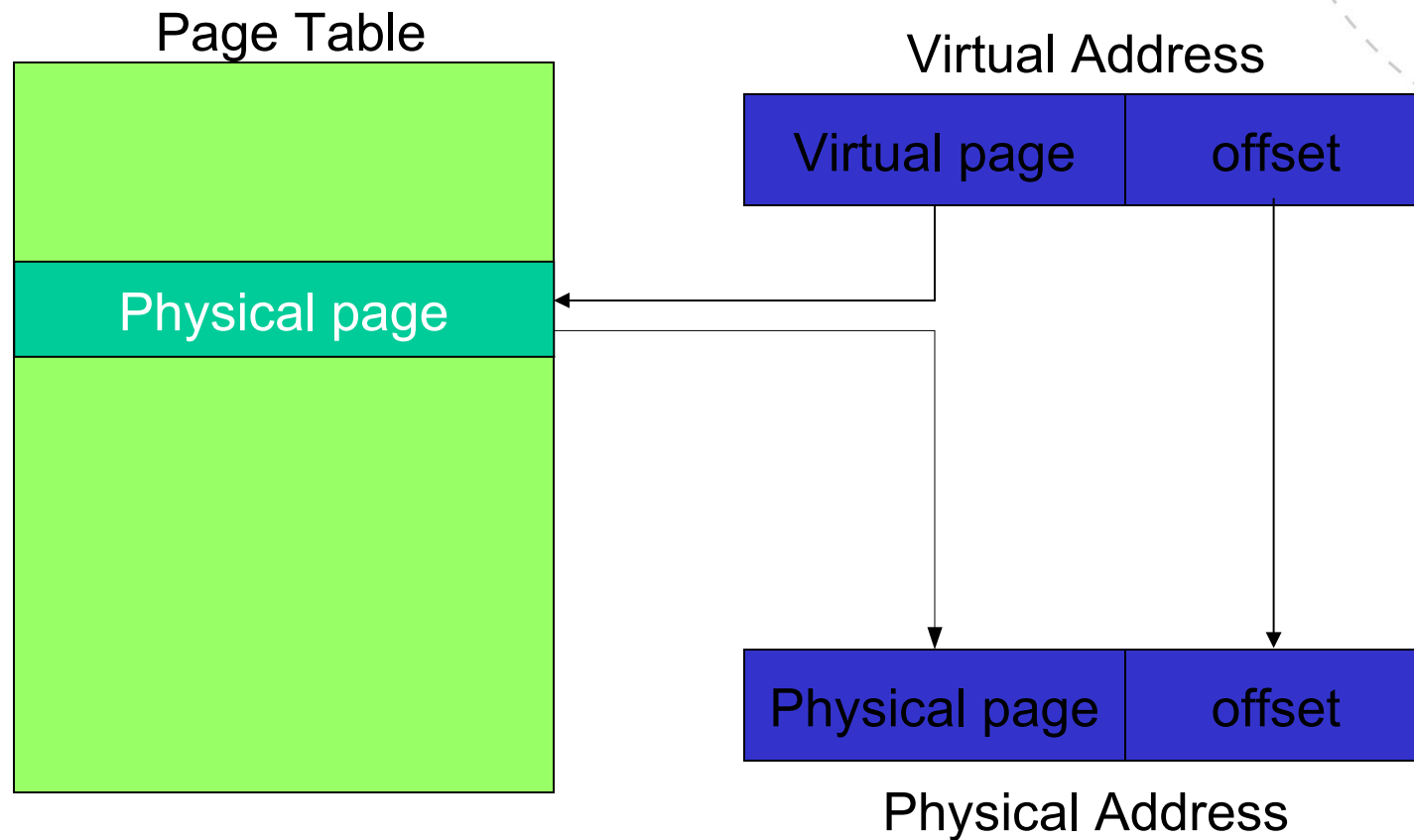
Virtual memory and paging

- Divide memory into fixed size blocks, called pages
 - Each process has its own virtual address space
 - A virtual page address can be mapped to any physical page
- Memory Management Unit (MMU)
 - Translates from virtual to physical addresses
 - Transparent for SW (except for setup)

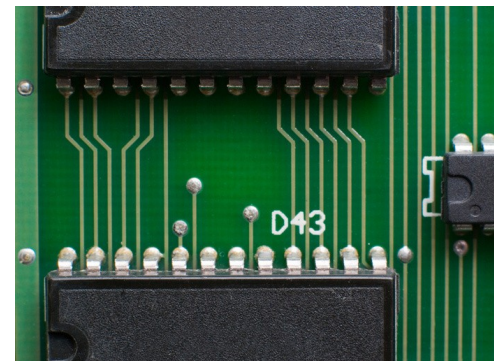
Memory

Page 1
Page 2
Page 3
Page 4
Page 5
Page 6
Page 7
Page 8

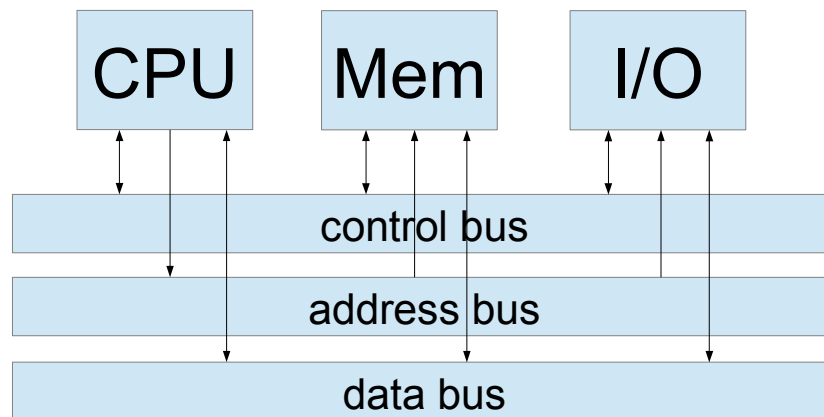
Page address translation



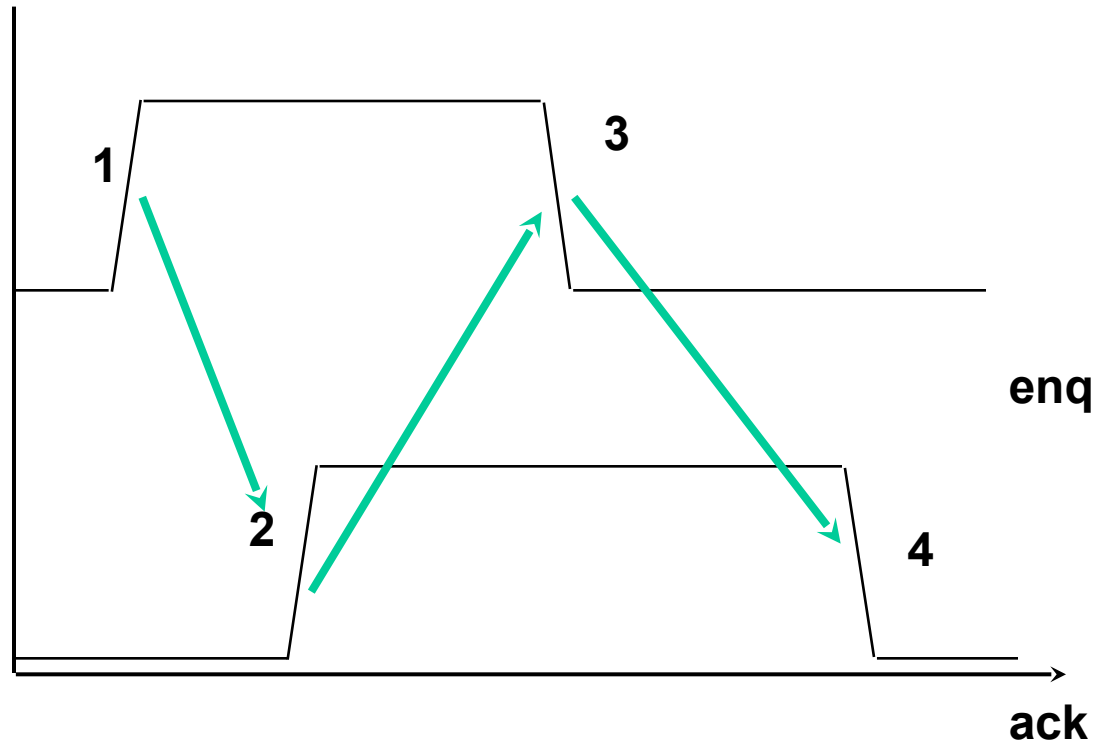
The bus



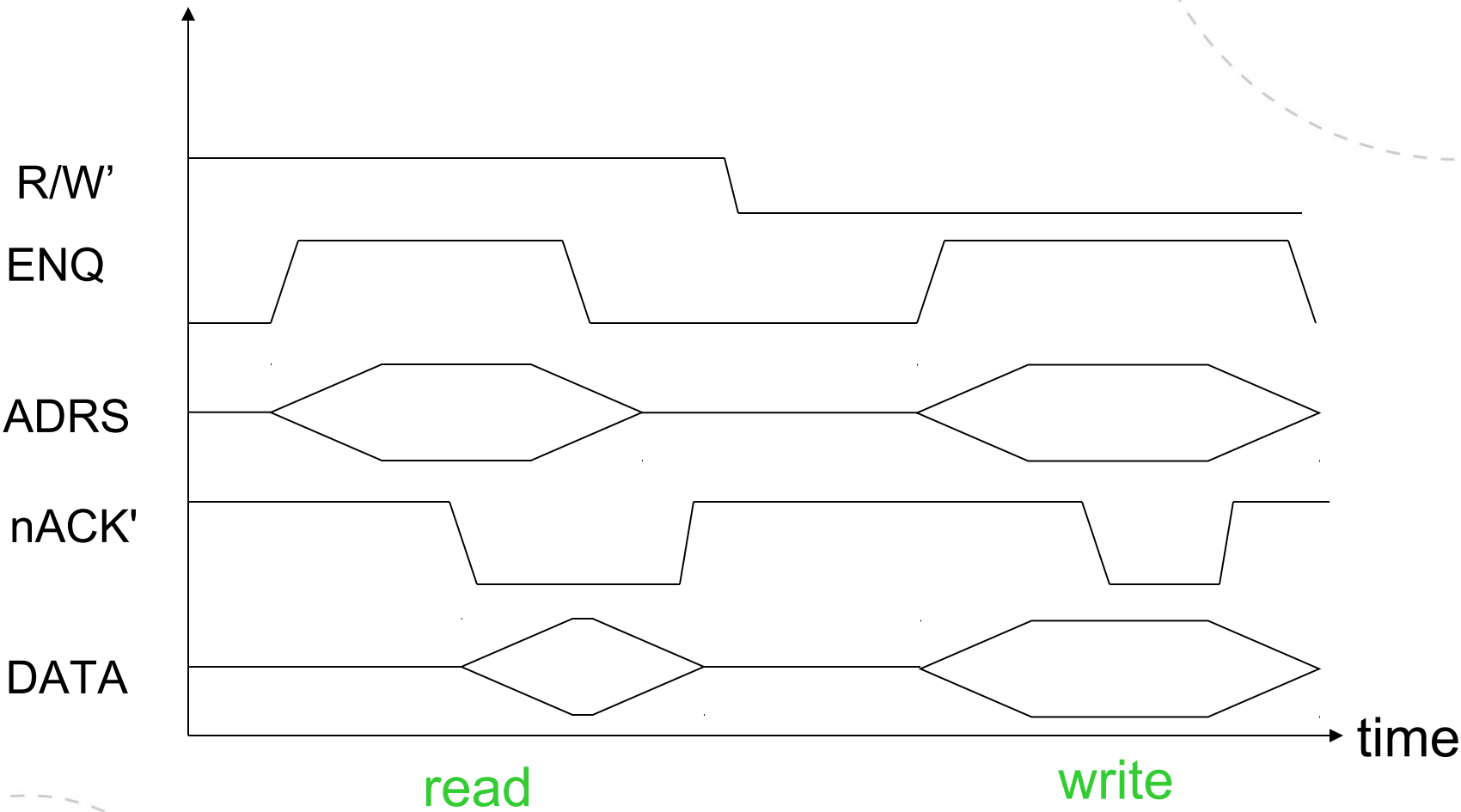
- A bus is a collection of wires
 - Provide mechanisms for data transfer, control and signaling
 - Multiplexing is possible
- The rules that governs the communication between two devices is called a protocol
- The master initiates transfers, slaves respond



Bus protocol: Four-cycle handshake

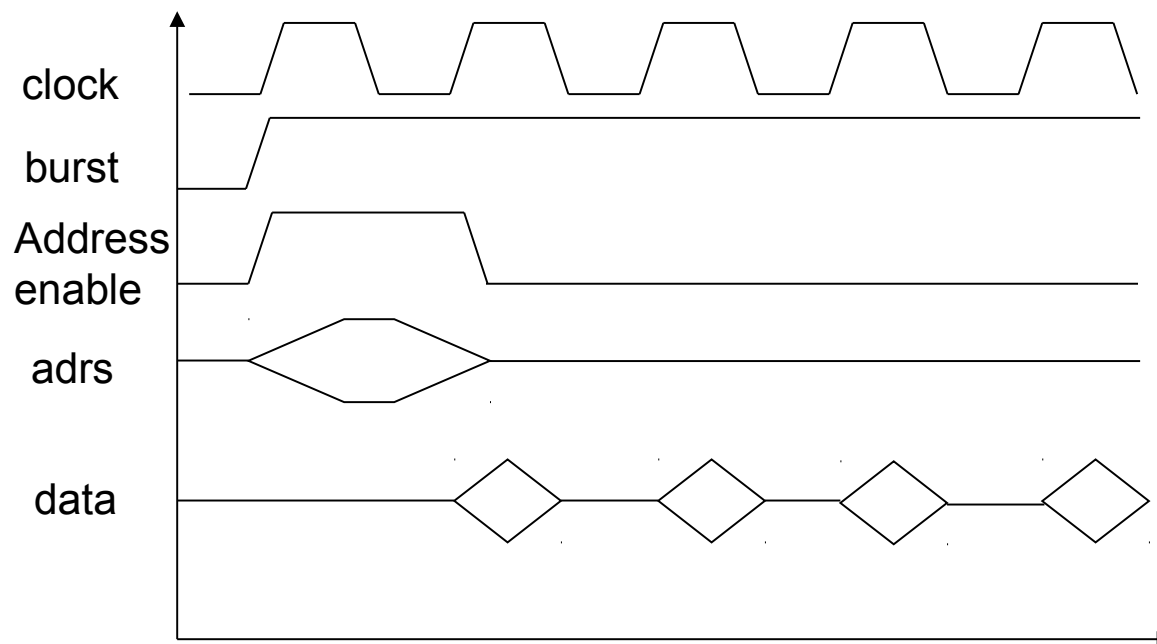


Typical asynchronous bus access

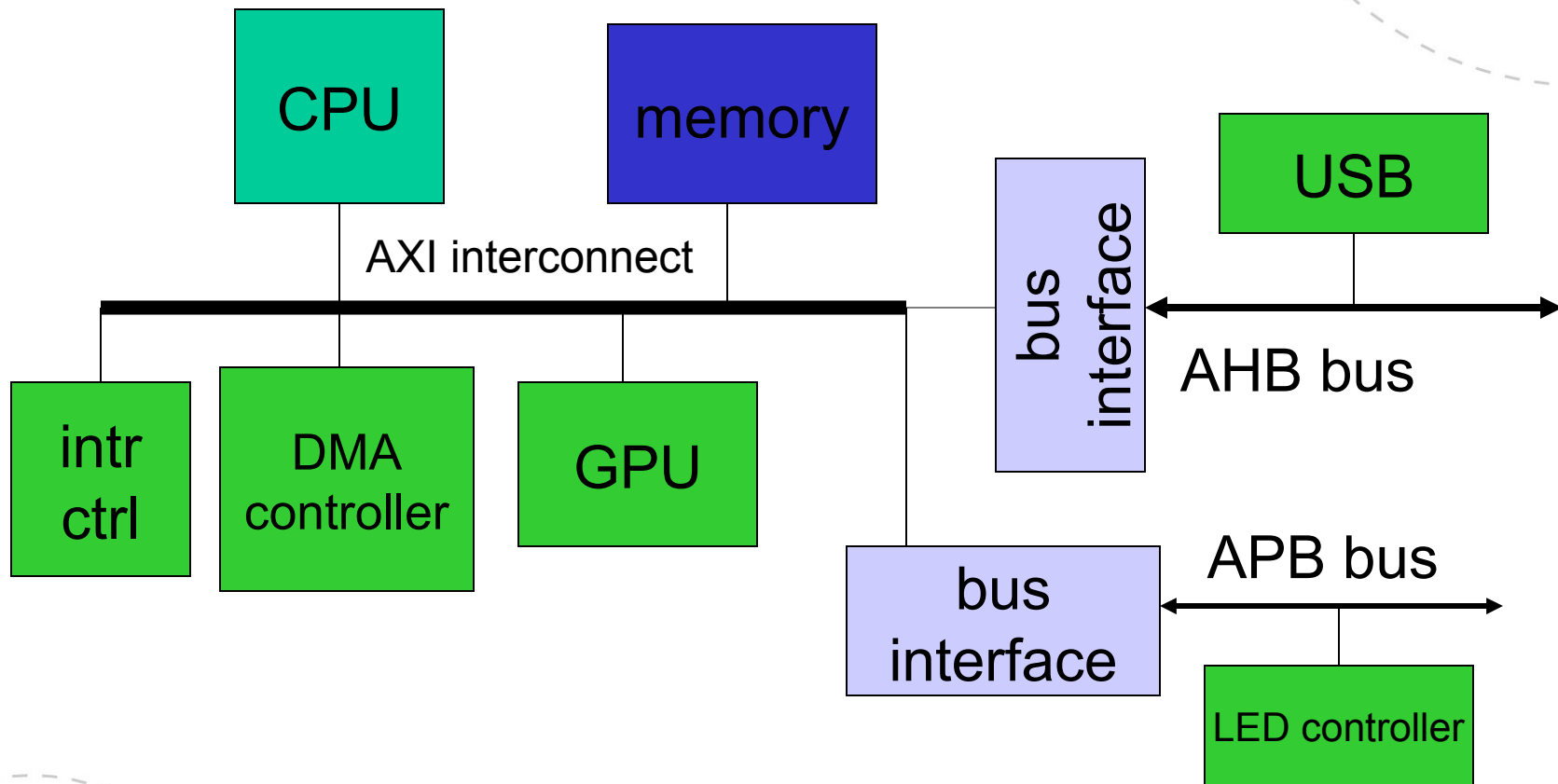


Synchronous burst transfer

- Some busses support transferring multiple words in one request
- Saves lots of time wasted on initiating requests multiple times

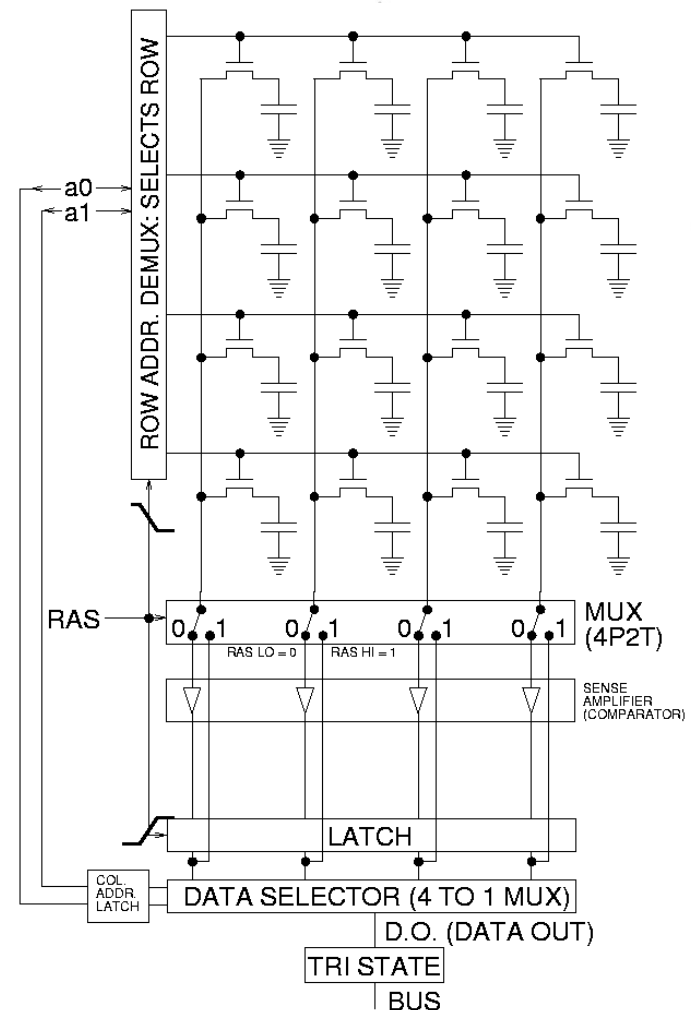


ARM bus hierarchy



DRAM array

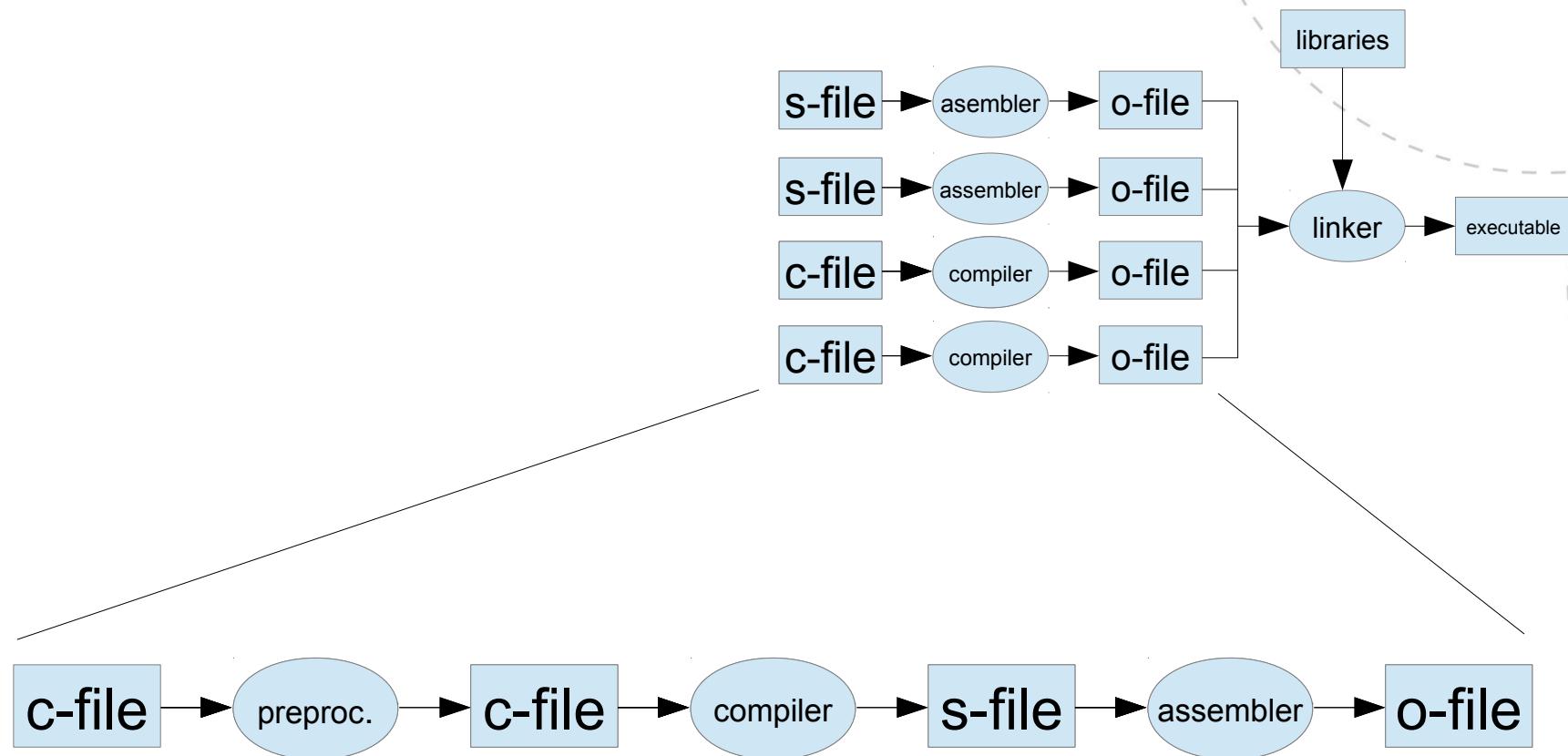
1. Precharge bitlines
2. Open row
 - Sense amps helps drive bitlines high or low
3. Latch in row
4. Mux out wanted column



Compiler tool chain

Chapter 5 Exercises

Compiler tool chain



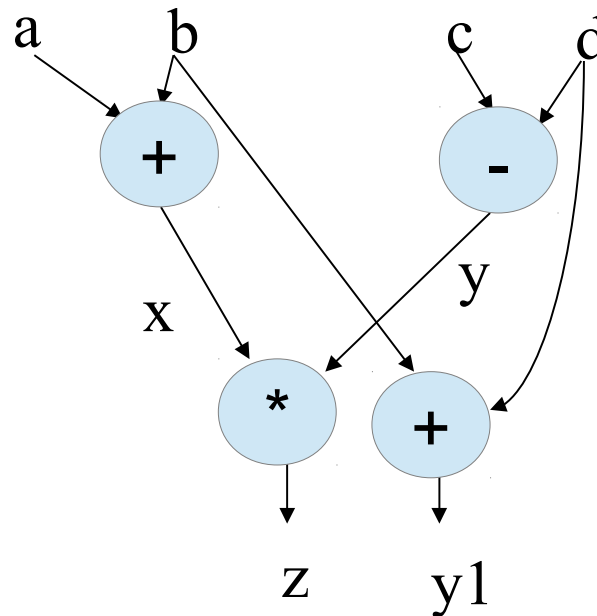
Data flow graph (DFG)

$x = a + b;$
 $y = c - d;$
 $z = x * y;$
 $y1 = b + d;$

Basic block in single assignment form

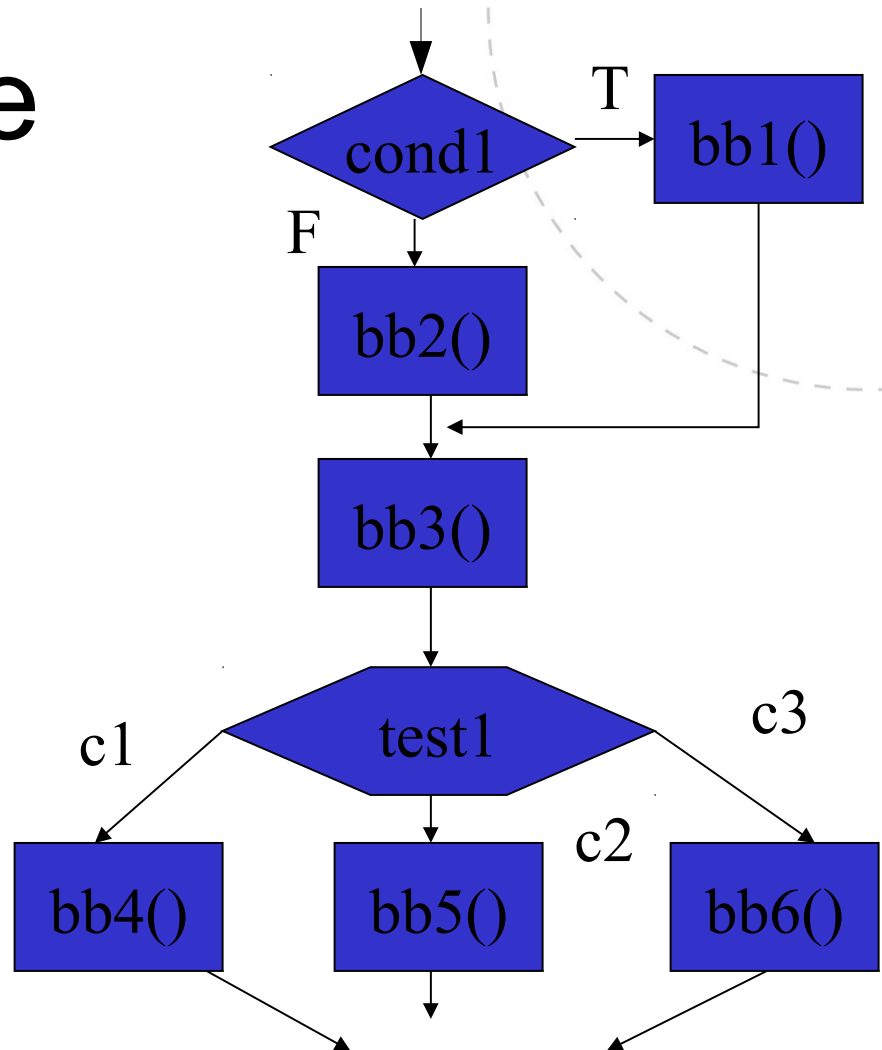
Partial orders:

1. $a+b$ and $c-d$
2. $x*y$ and $b+d$



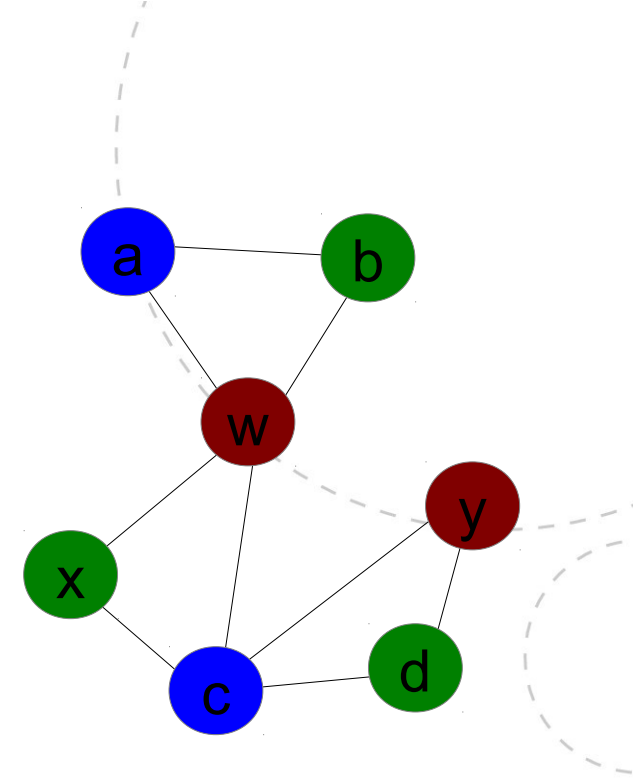
CDFG Example

```
if (cond1) {  
    bb1();  
} else {  
    bb2();  
}  
bb3();  
switch (test1) {  
    case c1: bb4(); break;  
    case c2: bb5(); break;  
    case c3: bb6(); break;  
}
```



Register allocation

- Method: Conflict graph and graph coloring
 - Edge between variables that are alive at the same time
 - Represent each register with a color
 - Color the nodes with as few colors as possible
 - No edge must share a color
 - NP-complete
 - Compilers use heuristics to find a good solution



3 registers

a r0
b r1
w r2
x r1
c r0
y r2
d r1

The object file, linking

- Object file: Result of assembly
- Several standards:
 - ELF (unix), COFF (unix, windows)
- The object file includes:
 - Symbol table
 - Binary program code (text segment)
 - Data (data segment)
 - Uninitialized data (bss segment)
 - Information about relocatable parts
 - Debug data (references to source files)

Embedded operating systems

Exercises Chapter 6

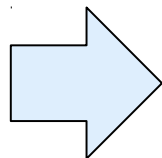
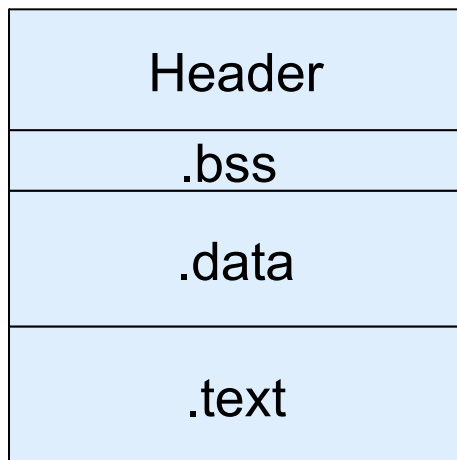
Making an OS

- You need startup code
 - Boot vectors
 - Exception handling
 - Stack and heap
 - Cache and MMU
 - Jump to main()
- You probably want support for libc
 - You must implement support for the syscalls required by libc.
 - Can then use malloc(), strcmp(), printf(), etc
- You probably need support for critical sections
 - Interrupt handlers communicating with main loop
 - mutex, semaphore
- A simple task scheduler can be useful

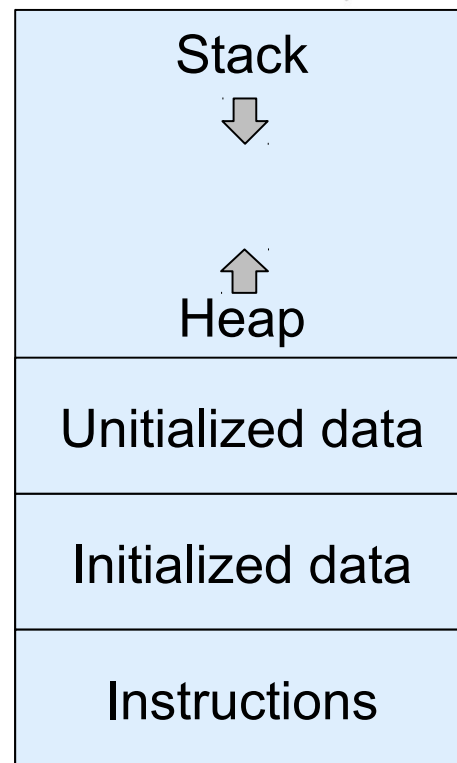
Process and thread

- A *process* is an instance of a program in memory
 - Instructions and state
 - Its own virtual address space
 - A *thread* shares memory with other threads in the same process
- Single-tasking operating systems only have one process in memory at a time
- Multitasking operating systems can have several processes in memory
 - All modern operating systems
 - The *scheduler* manages which process is running
- Each process has an associated data structure used by the process scheduler

Stored Program Binary



Process in memory

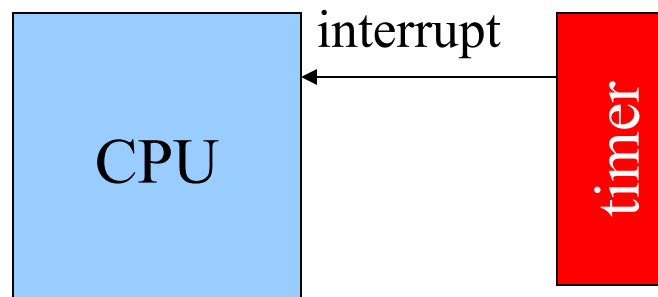


Context switching

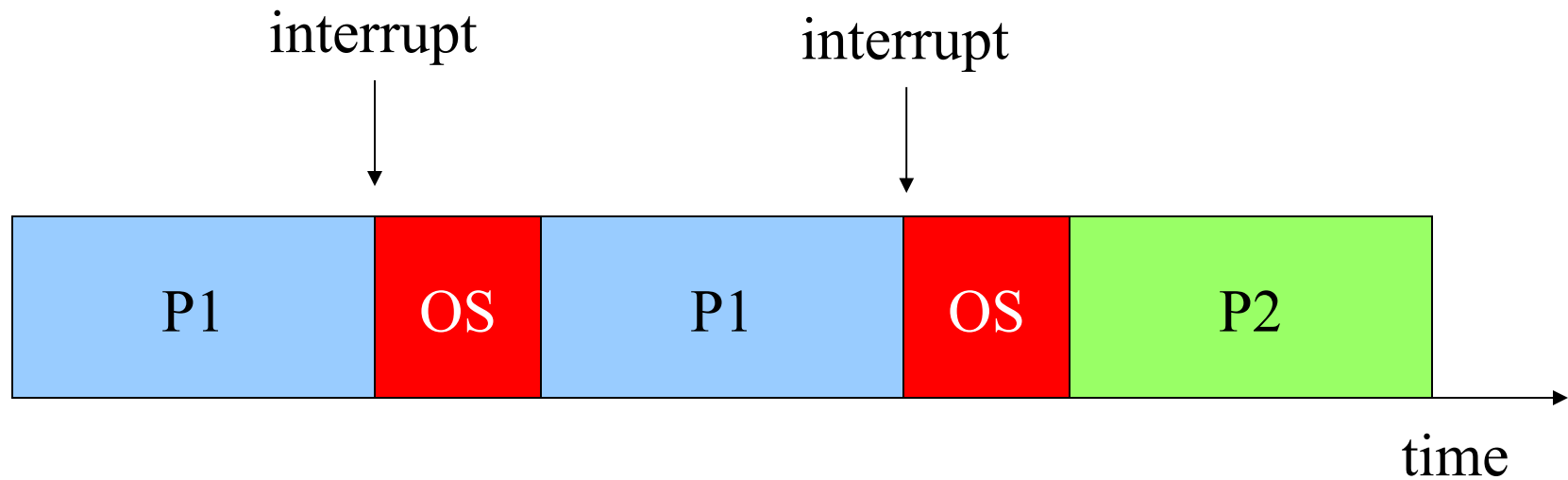
- Terms:
 - Context: The process state needed to be remembered when switching processes
 - Contains register state and kernel structures
 - Context switch:
 - Removes the running context and stores it
 - Inserts the new context
- Change which process is running on the CPU
- Implementation decisions:
 - Who decides when to switch?
 - How is the context switch implemented?
- Two variants:
 - Cooperative multitasking
 - Preemptive multitasking

Preemptive multitasking

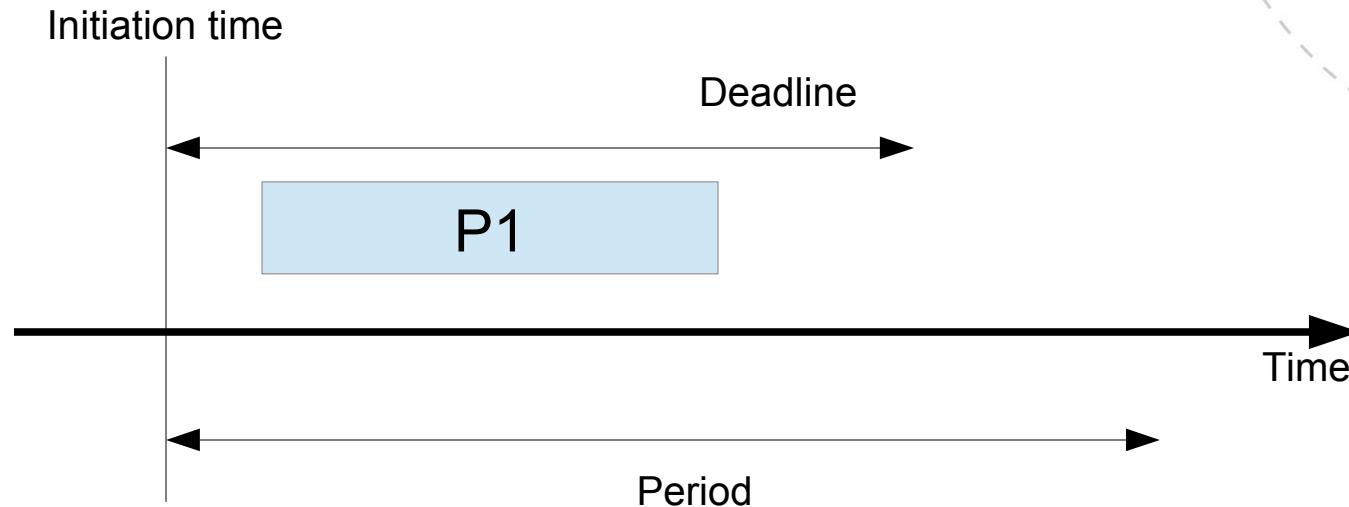
- OS decides when context switches are carried out and how processes are scheduled
- A timer interrupt gives control to the CPU at regular intervals



Preemption control flow



Real Time OS: Deadlines



- Initiation time
- Deadline
- Period

Synchronization mechanisms

- Critical section
 - Code accessing a shared resource that must not be interleaved with other threads accessing the same resource
 - Access to critical section must be atomic
 - Examples:
 - Accessing shared memory
 - Accessing an I/O device
- OS must provide semaphores or mutexes
 - Protect critical sections
- Protocol:
 - Lock the mutex with P()
 - Only one can have access
 - Execute operations on shared resource
 - Release the mutex with V()

Implementing mutex

- Need to atomically test and set a variable
- Single processor systems:
 - Possible to get atomicity by temporarily disabling interrupts
- Multi-processor systems: HW support
 - ARM: LDREX, STREX instruction pair

TAKEN = 0xFF

```
mov r1, #TAKEN
```

try:

```
ldrex r0, [LockAddr]
```

```
cmp r0, #0
```

```
strexeq r1, r0, [LockAddr]
```

```
cmpeq r0, #0
```

```
bne try
```

Energy and power

Chapters 3, 5, 6

Guest lectures

Power and Energy

- Dynamic power
 - Caused by switching transistors
 - Reduce by
 - Reducing voltage
 - Reducing transistor switching
- Static power
 - Caused by leaking transistors
 - Reduce by
 - Reducing voltage
- Power consumption depends on process technology
- Energy: Power consumption over time ($J = Ws$)

Static SW techniques for energy efficiency

- Memory access optimization
 - Memory transfers consumes a lot of energy
 - Especially if off-chip
 - Tens to hundreds of times more expensive than an ALU operation
 - Make sure registers are allocated properly to reduce loads and stores
 - Tune algorithms and datastructures such that your working set fits in cache
 - Minimize memory footprint
- I/O access optimization
 - Reduce flash/HD usage
 - Reduce network traffic

Static SW techniques for energy efficiency

- Overall advice: *High performance == low power*
- Execution time optimization
 - Clear correlation between execution time and energy consumption
 - All techniques for reducing execution time will likely reduce energy consumption roughly as much
- Make use of your available HW
 - Unused resources wastes energy
 - Use available resources to make execution time as short as possible
 - Example: If you have support for vector instructions (ARM NEON), their usage can probably help reduce energy consumption
 - Avoid using processors with capabilities you don't need

OS Power management

- Power management: Controlling system resources with the aim of reducing power consumption
- OS can prioritize for power consumption
 - Similarly as it does for execution time
- Main techniques:
 - Sleep modes
 - Run-time enabling/disabling devices
 - Frequency and voltage scaling

Sleep modes

- When the CPU is idle, it can be put to sleep to save energy (powered off)
- Several sleep modes are often available, with various compromises regarding energy consumption and wake-up time
- Typical variations:
 - Which clocks are running
 - Which I/O controllers are powered on
- OS must support this by entering appropriate sleep modes in the idle loop

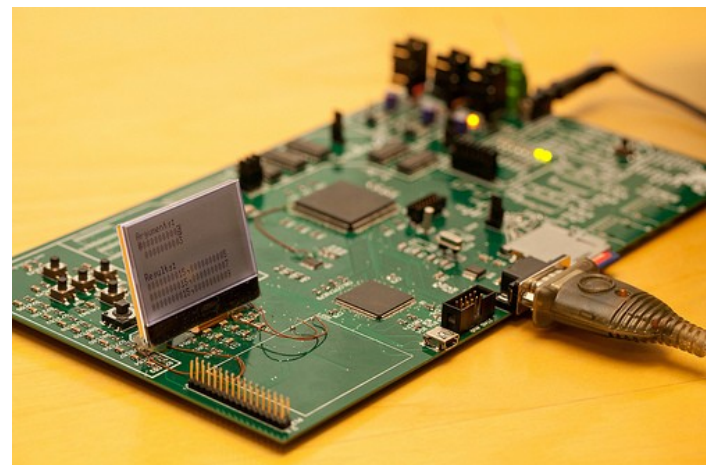
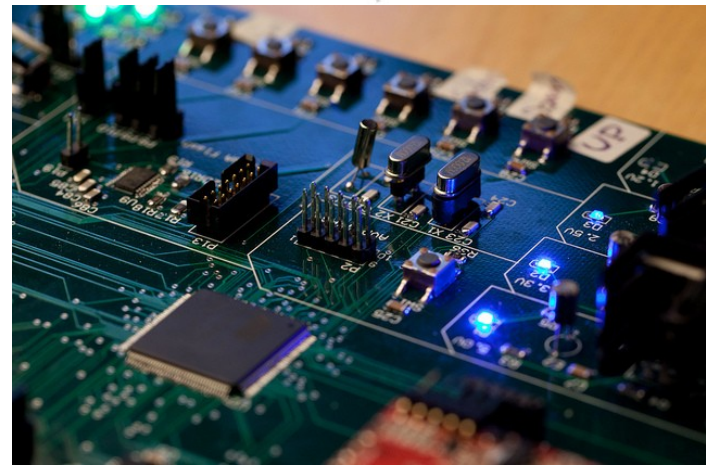
Dynamic voltage and frequency scaling (DVFS)

- Reducing clock frequency reduces dynamic power consumption
- Reducing voltage reduces both dynamic and static power consumption
- Given HW that can do DVFS, the OS should adjust both of these such that the CPU is running “just fast enough”.

More advanced courses

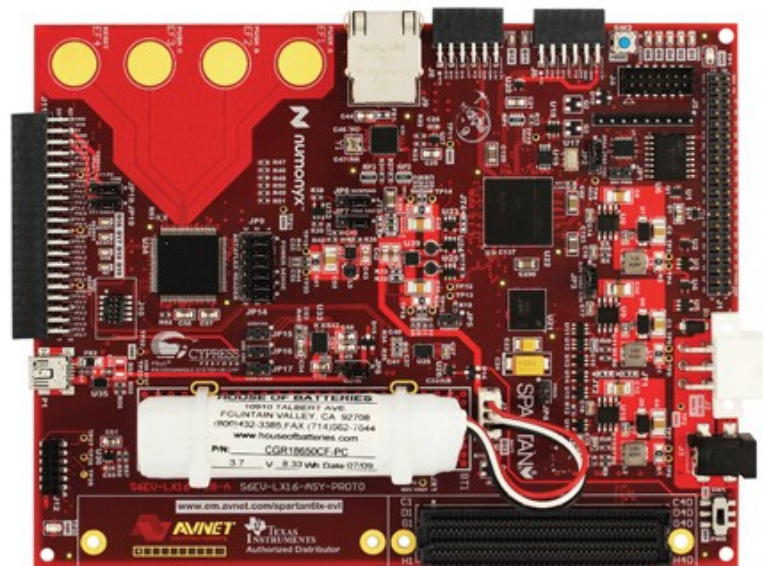
TDT4295 Computer Design Project

- Assignment: Build exotic computer system
- Tasks:
 - PCB design
 - Custom processor in an FPGA
 - AVR for I/O
- One group of roughly 10 students
- Duration: 12 weeks



TDT4255 Computer Design

- Teaches scalar processor core design
- Exam counts 50%, exercises count 50%
- Students design, implement and verify 2 different processor designs



Xilinx Spartan6 Lx16 Evaluation Kit

TDT4260 Computer Architecture

- Teaches how high-level building blocks are assembled to a complete computer system
- Exam counts 80%, exercises count 20%
- Hands-on experience with simulator-based architecture analysis
 - GEM5 simulator (C++, cycle accurate)
 - Task: Build the best possible prefetcher
 - Continuous evaluation of the best student prefetcher



Good luck