F28HS Hardware-Software Interface: Systems Programming

Hans-Wolfgang Loidl

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

Semester 2 — 2018/19

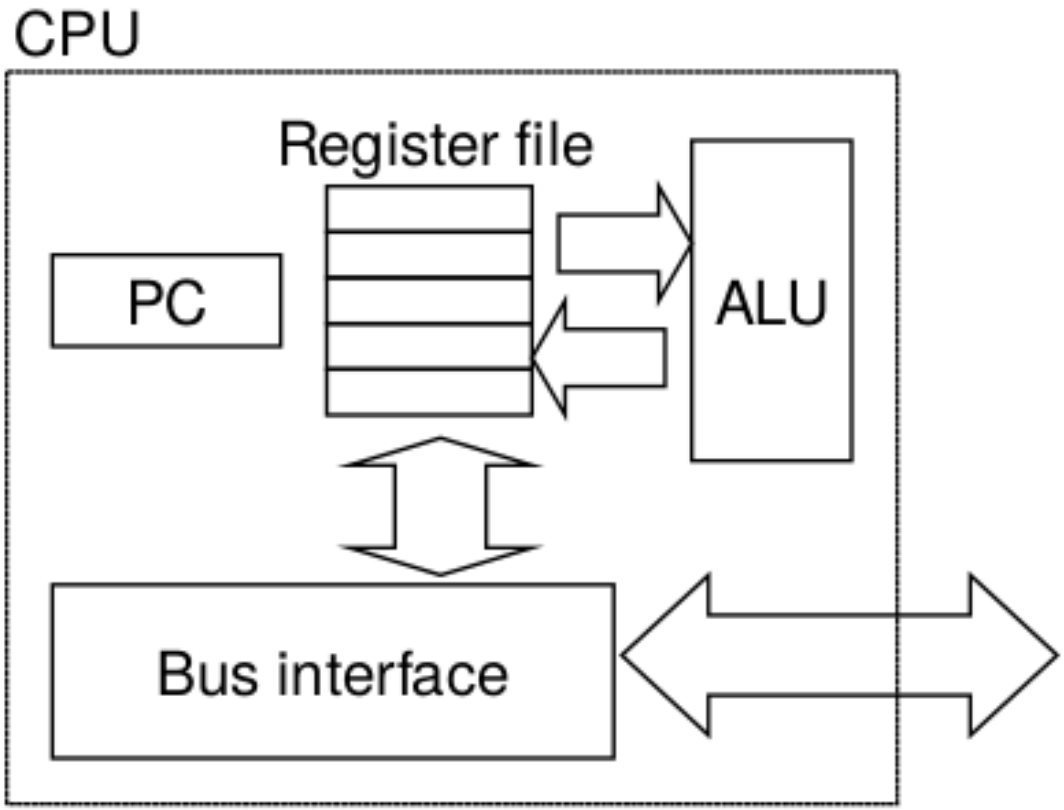
[0]No proprietary software has been used in producing these slides

Outline

HERIOT WATT UNIVERSITY

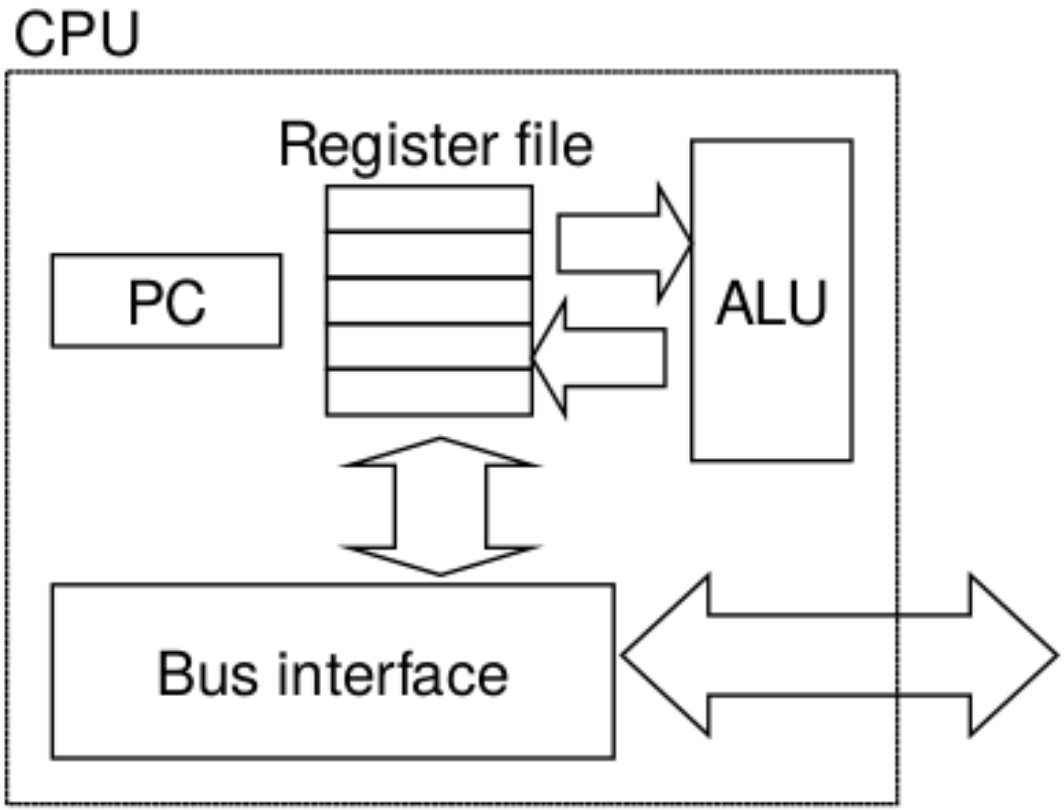
Lecture 10: Revision

A simple picture of the CPU



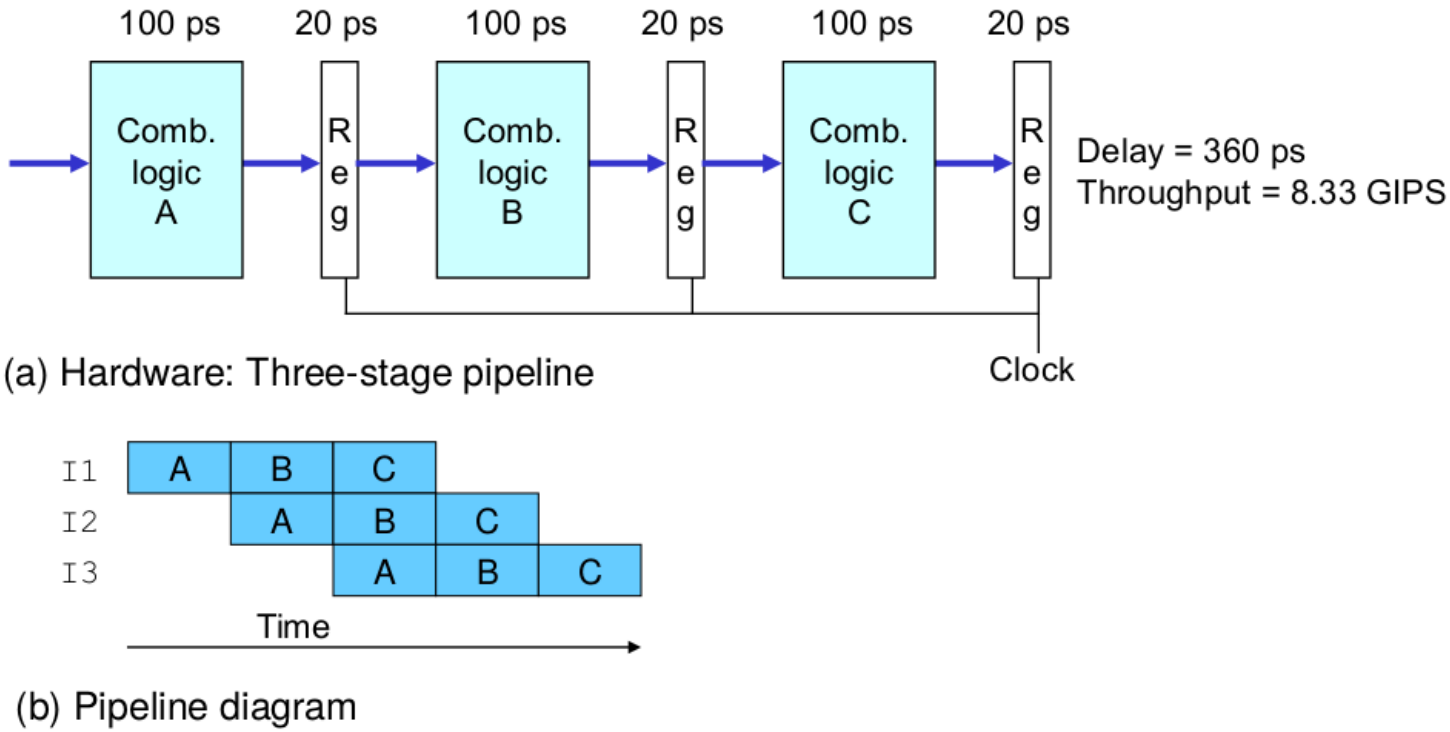
- The ALU executes arithmetic/logic operations with arguments in registers
- Load and store instructions move data between memory and registers

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

Three-stage pipelined computation hardware



The computation is split into stages A, B, and C. The stages for different instructions can be executed in an overlapping way.

[0] From Bryant, Chapter 4

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.

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

HERIOT WATT UNIVERSITY

Example: bad branch prediction

```
          .global _start
          .text
_start:   MOVS  R1, #0      @ load 0 =>
          LSR   R1, #1      @ LSR yields zero =>
          BNE   target      @ Not taken
          MOV   R0, #0      @ fall through
          MOV   R7, #1
          SWI   0
target:   MOV   R0, #1      @ return: branch taken?
          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 **NOT** be taken, because the right shift (`LSR`) will set the zero flag according to the right-most bit, which is **0** in this case. This is a deliberately **bad** example for the branch

Example: good branch prediction

```
          .text
_start:   MOVS  R1, #1      @ load 1 =>
          LSR   R1, #1      @ LSR yields one =>
          BNE   target      @ Branch taken
          MOV   R0, #0      @ fall through
          MOV   R7, #1
          SWI   0
target:   MOV   R0, #1      @ return: branch taken?
          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: now the conditional branch (BNE) **WILL** be taken, because the right shift (`LSR`) will set the zero flag according to the right-most bit, which is **1** in this case. This is better for the branch predictor and gives better performance.

Performance: good vs bad branch prediction

We now measure the performance of doing these two versions inside two nested loops (`0x10000` iterations, each).
Good Case: branch taken:

```
> hawopi[167](4.2)> as -o bonzo15.o bonzo15.s
> hawopi[168](4.2)> ld -o bonzo15 bonzo15.o
> hawopi[169](4.2)> time ./bonzo15
 real    0m30.091s
 user    0m29.980s
 sys     0m0.000s
> hawopi[170](4.2)> echo $?
 1
```

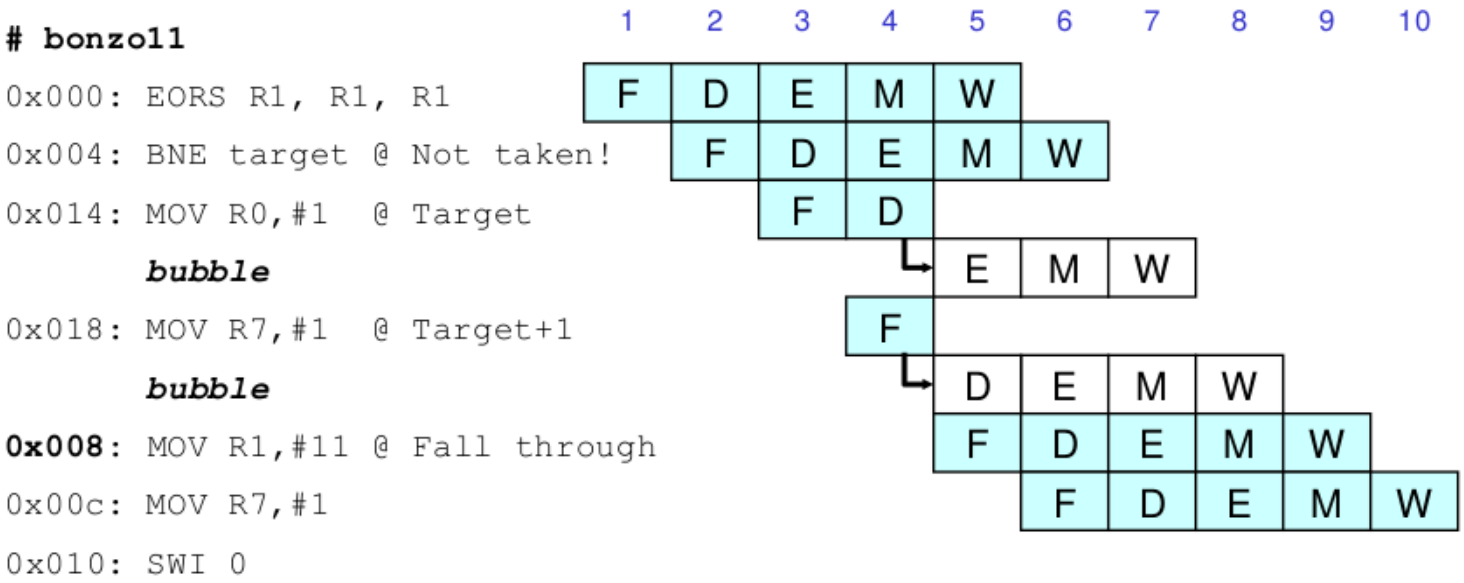
Performance: good vs bad branch prediction

We now measure the performance of doing these two versions inside two nested loops (`0x10000` iterations, each).
Bad Case: branch NOT taken:

```
> hawopi[171](4.2)> as -o bonzo15.o bonzo15.s
> hawopi[172](4.2)> ld -o bonzo15 bonzo15.o
> hawopi[173](4.2)> time ./bonzo15
 real    0m36.188s
 user    0m34.900s
 sys     0m0.090s
> hawopi[174](4.2)> echo $?
 0
```

NB: a difference in runtime of ca. 16.8%

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

The Current Program Status Register (CPSR)

The Current Program Status Register (CPSR) contains flags (`V,Z,N,C`) that are set by certain assembler instructions.
For example, the `CMP R0, R1` instruction compares the values of registers `R0` and `R1` and sets the zero flag (`Z`) if `R0` and `R1` are equal.

Figure 3-6 shows the bit assignments in the CPSR.

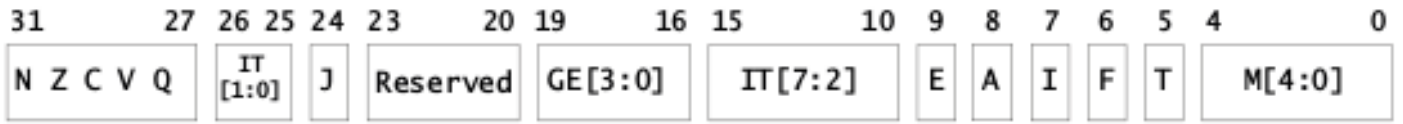
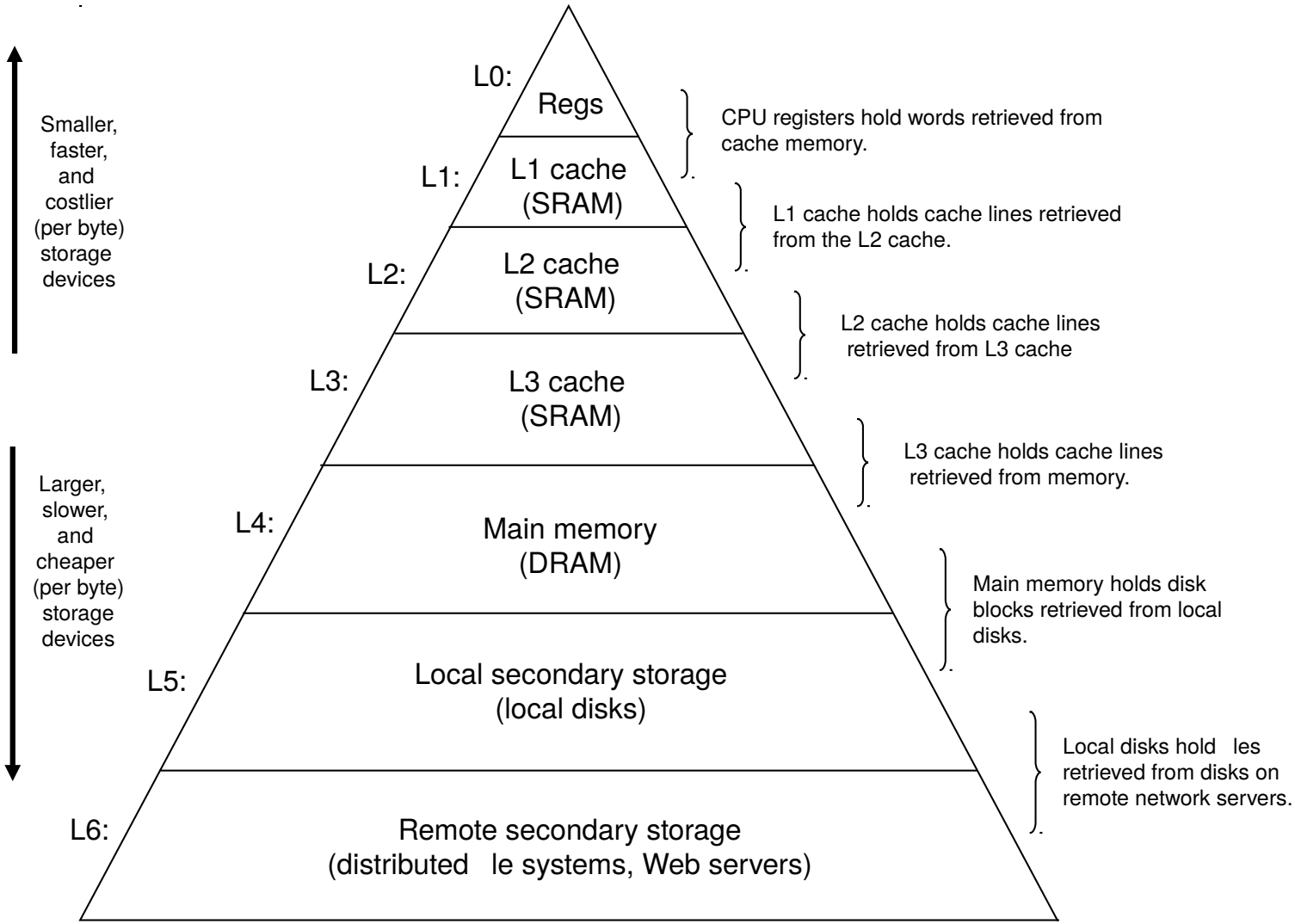


Figure 3-6 CPSR bits

[0] See ARM's Programmer Guide, p. 3-8

Caches and Memory Hierarchy



Discussion

As we move from the top of the hierarchy to the bottom, the devices become **slower, larger, and less costly** per byte.

The main idea of a memory hierarchy is that **storage at one level serves as a cache for storage at the next lower level**.

Using the different levels of the memory hierarchy efficiently is crucial to achieving high performance.

Access to levels in the hierarchy can be explicit (for example when using OpenCL to program a graphics card), or implicit (in most other cases).

Importance of Locality

Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer!

Which of the following two version of sum-over-matrix has better locality (and performance):

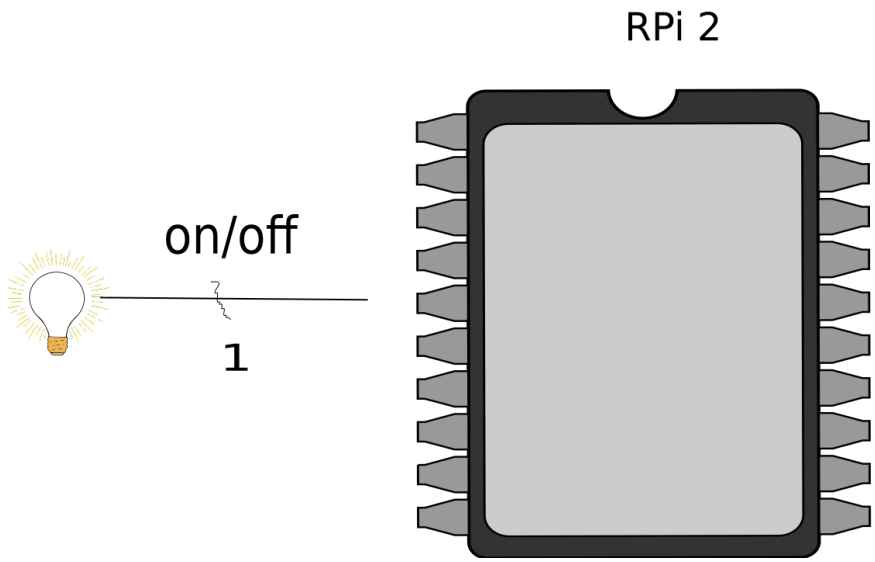
Traversal by rows:

```
int i, j;  ulong  sum;
for (i = 0; i<n; i++)
  for (j = 0; j<n; j++)
    sum += arr[i][j];
```

Traversal by columns:

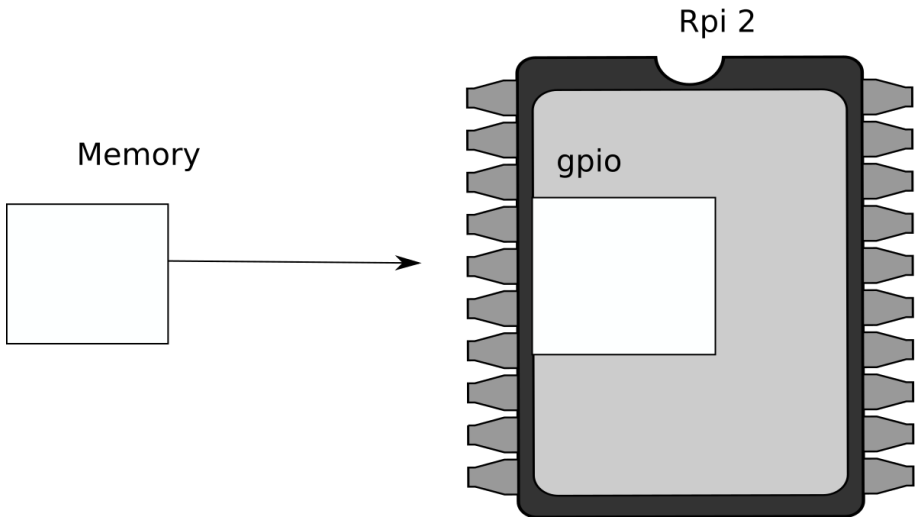
```
int i, j; ulong  sum;
for (j = 0; j<n; j++)
  for (i = 0; i<n; i++)
  sum += arr[i][j];
```

The high-level picture



- From the main chip of the RPi2 we want to control an (external) device, here an LED.
- We use one of the **GPIO** pins to connect the device.
- Logically we want to send **1 bit** to this device to turn it **on/off**.

The low-level picture



Programmatically we achieve that, by

- memory-mapping the address space of the GPIOs into user-space
- now, we can directly access the device via memory read/writes
- we need to pick-up the meaning of the peripheral registers from the BCM2835 peripherals sheet

BCM2835 GPIO Peripherals

Base adress: 0x3F000000

0	GPFSEL	Pins 0-9	(3-bits per pin)
5		Pins 50-53	
7	GPSET	Pins 0-31	(1-bit per pin)
8		Pins 32-53	
10	GPCLR	Pins 0-31	(1-bit per pin)
11		Pins 32-53	
13	GPLEV	Pins 0-31	(1-bit per pin)
14		Pins 32-53	
...			

The meaning of the registers is (see p90ff of BCM2835 ARM peripherals):

- **GPFSEL**: function select registers (3 bits per pin); set it to 0 for input, 1 for output; 6 more alternate functions available
- **GPSET**: **set** the corresponding pin
- **GPCLR**: **clear** the corresponding pin
- **GPLEV**: return the **value** of the corresponding pin

GPIO Register Assignment

Address	Field Name	Description	Size	Read/ Write
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0004	GPFSEL1	GPIO Function Select 1	32	R/W
0x 7E20 0008	GPFSEL2	GPIO Function Select 2	32	R/W
0x 7E20 000C	GPFSEL3	GPIO Function Select 3	32	R/W
0x 7E20 0010	GPFSEL4	GPIO Function Select 4	32	R/W
0x 7E20 0014	GPFSEL5	GPIO Function Select 5	32	R/W
0x 7E20 0018	-	Reserved	-	-
0x 7E20 001C	GPSET0	GPIO Pin Output Set 0	32	W
0x 7E20 0020	GPSET1	GPIO Pin Output Set 1	32	W
0x 7E20 0024	-	Reserved	-	-
0x 7E20 0028	GPCLR0	GPIO Pin Output Clear 0	32	W
0x 7E20 002C	GPCLR1	GPIO Pin Output Clear 1	32	W
0x 7E20 0030	-	Reserved	-	-

The GPIO has 48 32-bit registers (RPi2; 41 for RPi1).

[0]See BCM Peripherals Manual, Chapter 6, Table 6.1

GPIO Register Assignment

GPIO registers (Base address: 0x3F200000)

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
GPFSEL0	0:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
GPFSEL1	1:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
GPFSEL2	2:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
GPFSEL3	3:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
GPFSEL4	4:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
GPFSEL5	5:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
—	6:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
GPFSET0	7:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
GPFSET1	8:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
—	9:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
GPFCLR0	10:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
GPFCLR1	11:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13
—	12:	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13

[0] See BCM Peripherals, Chapter 6, Table 6.1

Locating the GPFSEL register for pin 47 (ACT)

Bit(s)	Field Name	Description	Type	Reset
31-30	---	Reserved	R	0
29-27	FSEL49	FSEL49 - Function Select 49 000 = GPIO Pin 49 is an input 001 = GPIO Pin 49 is an output 100 = GPIO Pin 49 takes alternate function 0 101 = GPIO Pin 49 takes alternate function 1 110 = GPIO Pin 49 takes alternate function 2 111 = GPIO Pin 49 takes alternate function 3 011 = GPIO Pin 49 takes alternate function 4 010 = GPIO Pin 49 takes alternate function 5	R/W	0
26-24	FSEL48	FSEL48 - Function Select 48	R/W	0
23-21	FSEL47	FSEL47 - Function Select 47	R/W	0
20-18	FSEL46	FSEL46 - Function Select 46	R/W	0
17-15	FSEL45	FSEL45 - Function Select 45	R/W	0
14-12	FSEL44	FSEL44 - Function Select 44	R/W	0
11-9	FSEL43	FSEL43 - Function Select 43	R/W	0
8-6	FSEL42	FSEL42 - Function Select 42	R/W	0
5-3	FSEL41	FSEL41 - Function Select 41	R/W	0
2-0	FSEL40	FSEL40 - Function Select 40	R/W	0

Table 6-6 – GPIO Alternate function select register 4

HERIOT WATT UNIVERSITY

Accessing a GPIO Pin

- Now we want to control the on-chip LED, called ACT, that normally indicates activity.
- The pin number of this device on the RPi2 is: **47**
- We need to calculate registers and bits corresponding to this pin
- The **GPFSEL** register for pin 47 is **4** (per docu, this register covers pins 40-49 (Tab 6-6, p. 94)
- For each register 3 bits are used to select the function of that pin: bits 0–2 for register 40 etc
- Thus, bits 21–23 cover register 47 (7×3)
- The function that we need to select is OUTPUT, which is encoded as the value 1
- We need to write the value `0x01` into bits 21–23 of register 4

Accessing a GPIO Pin

- Now we want to control the on-chip LED, called ACT, that normally indicates activity.
- The pin number of this device on the RPi2 is: **47**
- We need to calculate registers and bits corresponding to this pin
- The **GPFSEL** register for pin 47 is **4** (per docu, this register covers pins 40-49 (Tab 6-6, p. 94)
- For each register 3 bits are used to select the function of that pin: bits 0–2 for register 40 etc
- Thus, bits 21–23 cover register 47 (7×3)
- The function that we need to select is OUTPUT, which is encoded as the value 1
- We need to write the value `0x01` into bits 21–23 of register 4

Accessing a GPIO Pin

- Now we want to control the on-chip LED, called ACT, that normally indicates activity.
- The pin number of this device on the RPi2 is: **47**
- We need to calculate registers and bits corresponding to this pin
- The **GPFSEL** register for pin 47 is **4** (per docu, this register covers pins 40-49 (Tab 6-6, p. 94)
- For each register 3 bits are used to select the function of that pin: bits 0–2 for register 40 etc
- Thus, bits 21–23 cover register 47 (7×3)
- The function that we need to select is OUTPUT, which is encoded as the value 1
- We need to write the value `0x01` into bits 21–23 of register 4

HERIOT WATT UNIVERSITY

Accessing a GPIO Pin

- Now we want to control the on-chip LED, called ACT, that normally indicates activity.
- The pin number of this device on the RPi2 is: **47**
- We need to calculate registers and bits corresponding to this pin
- The **GPFSEL** register for pin 47 is **4** (per docu, this register covers pins 40-49 (Tab 6-6, p. 94)
- For each register 3 bits are used to select the function of that pin: bits 0–2 for register 40 etc
- Thus, bits 21–23 cover register 47 (7×3)
- The function that we need to select is OUTPUT, which is encoded as the value 1
- We need to write the value `0x01` into bits 21–23 of register 4

HERIOT WATT UNIVERSITY

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?
- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

HERIOT WATT UNIVERSITY

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?
- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

HERIOT WATT UNIVERSITY

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`? **Answer:** `gpio+4`
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?
- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

HERIOT WATT UNIVERSITY

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?
- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

HERIOT WATT UNIVERSITY

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
  Answer: `*(gpio+4)`
- How do we blank out bits 21–23 from this register?
- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

HERIOT WATT UNIVERSITY

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?
- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

HERIOT WATT UNIVERSITY

Accessing GPIO Pin 47

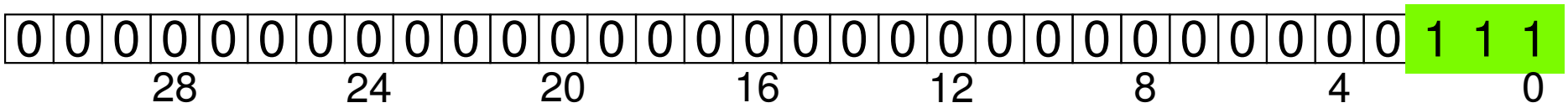
- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?
  Answer: `*(gpio + 4) & ~(7 << 21)`
- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

HERIOT WATT UNIVERSITY

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?

  C code: `7`

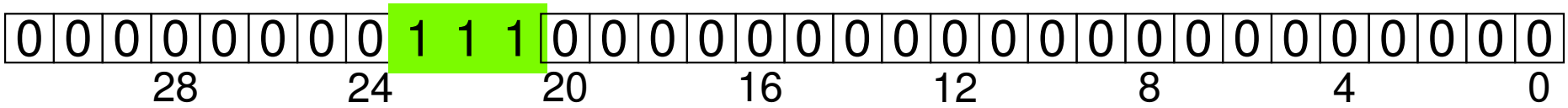


- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?

  C code: `7 << 21`

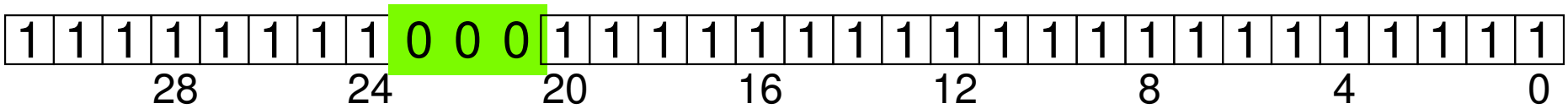


- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?

C code: `~(7 << 21)`

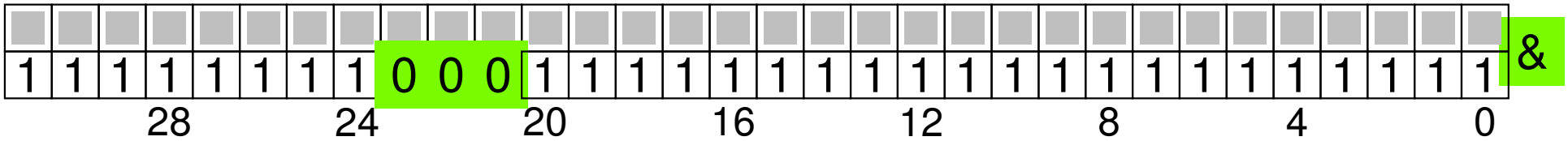


- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?

C code: `(*(gpio + 4) & ~(7 << 21))`

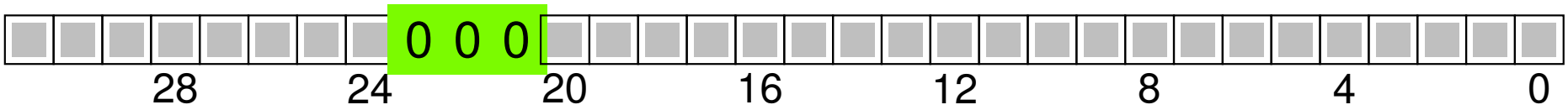


- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?

C code: `(*(gpio + 4) & ~(7 << 21))`



- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?
- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

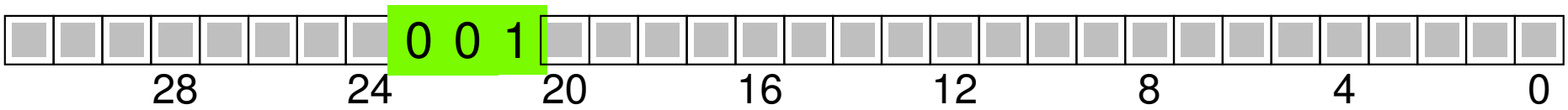
Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?
- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
  Answer: `(1 << 21)`
- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?
- How do we get the value `0x01` into bits 21–23 of a 32-bit word?

```
(*(gpio + 4) & ~(7 << 21)) | (1 << 21)
```



- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?
- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

Accessing GPIO Pin 47

- We want to construct C code to write the value `0x01` into bits 21–23 of register 4
- What's the address of register 4 relative to the base address in `gpio`?
- How do we read the current value from this register?
- How do we blank out bits 21–23 from this register?
- How do we get the value `0x01` into bits 21–23 of a 32-bit word?
- How do we put **only these bits** into the contents of register 4?

```
*(gpio + 4) = (*(gpio + 4) & ~(7 << 21)) | (1 << 21)
```

HERIOT WATT UNIVERSITY

GPIO programming

- The previous slides discussed how to control an LED with a GPIO pin.
- Similar code is used to use a button as an input device, and to read a bit from the right GPIO pin
- For the exam you need to understand the main steps that are needed
- You must be able to perform the above steps to explain, e.g. how to set the mode of a pin
- The LCD device is controlled in a similar way, but always sending 8 bits as the byte to be displayed.
- You should expect specific code questions about GPIO programming, either in C or Assembler

HERIOT WATT UNIVERSITY

Summary

- Check the detailed tutorial slides about controlling external devices
- Look-up the sample sources (both C and Asm) for the tutorials
- You need to have a **solid understanding of this code** and be able to answer questions about it!
- Focus on the main concepts that we covered in the lectures:
  - Computer architecture, in particular pipelining
  - Memory hierarchy, in particular caching
- You need to be able to explain how these concepts impact performance of some sample programs.
- Be prepared for **small-scale coding questions**

HERIOT WATT UNIVERSITY