NTNU
Norwegian University of
Science and Technology

Lecture 2: Instruction Sets and Assembly

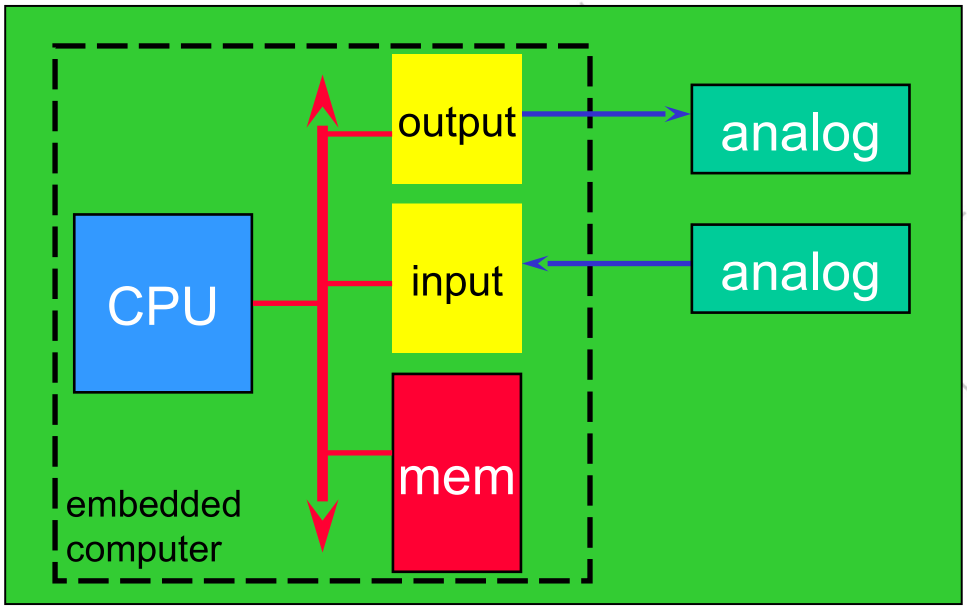
Asbjørn Djupdal
ARM Norway, IDI NTNU
2013

Lecture overview

- Introduction to processors and instruction sets
- Assembly language
- ARM architecture and instructions
- From C to assembly

Embedded Computer

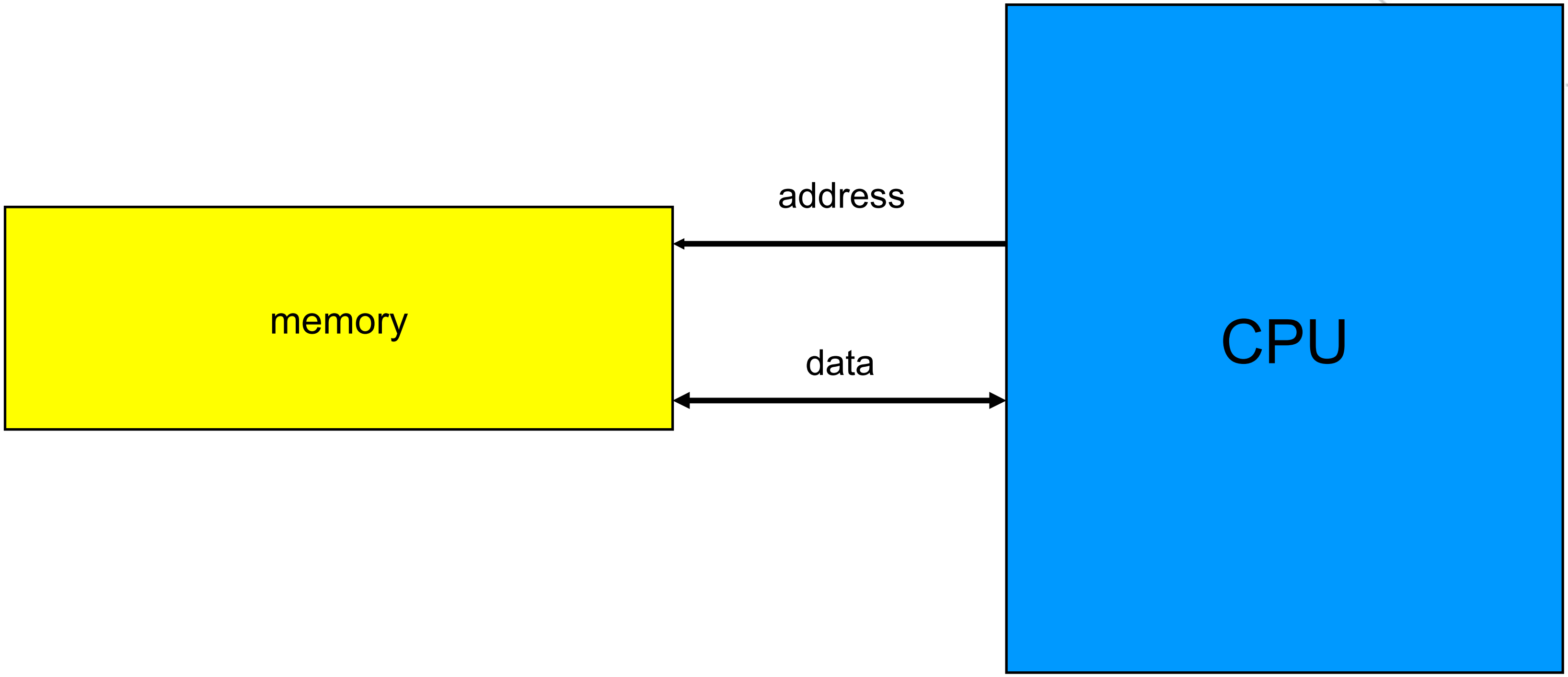
- Special purpose computer built into a bigger system
  - Digital clock
  - Washing machine
  - TV
- The most important component is the CPU
  - Todays topic



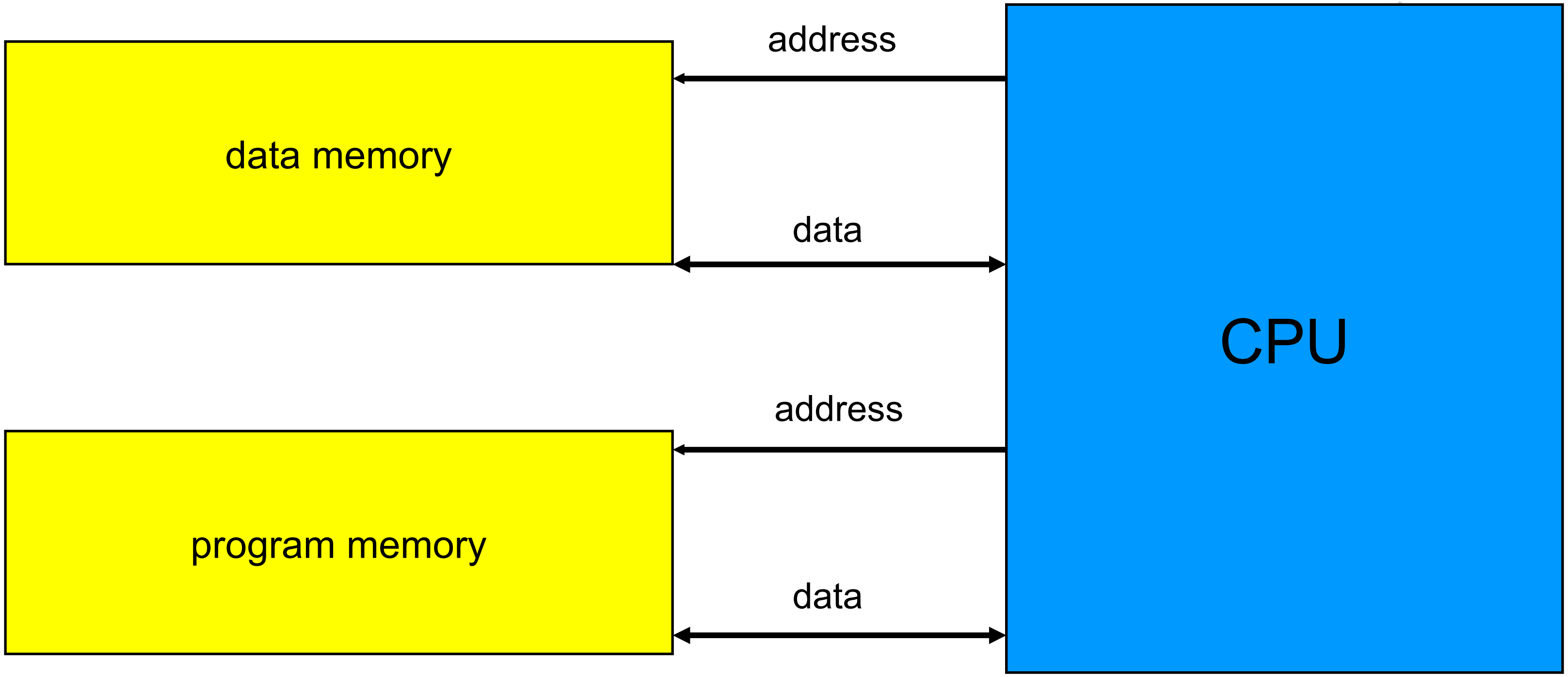
Computer Architecture

- High level organization of the computer
- Micro-Architecture: Organization of the processor core
- Two main architecture types:
  - Von Neumann (single memory)
  - Harvard (separate data and program memory)

von Neumann Architecture



Harvard Architecture



Von Neumann vs. Harvard

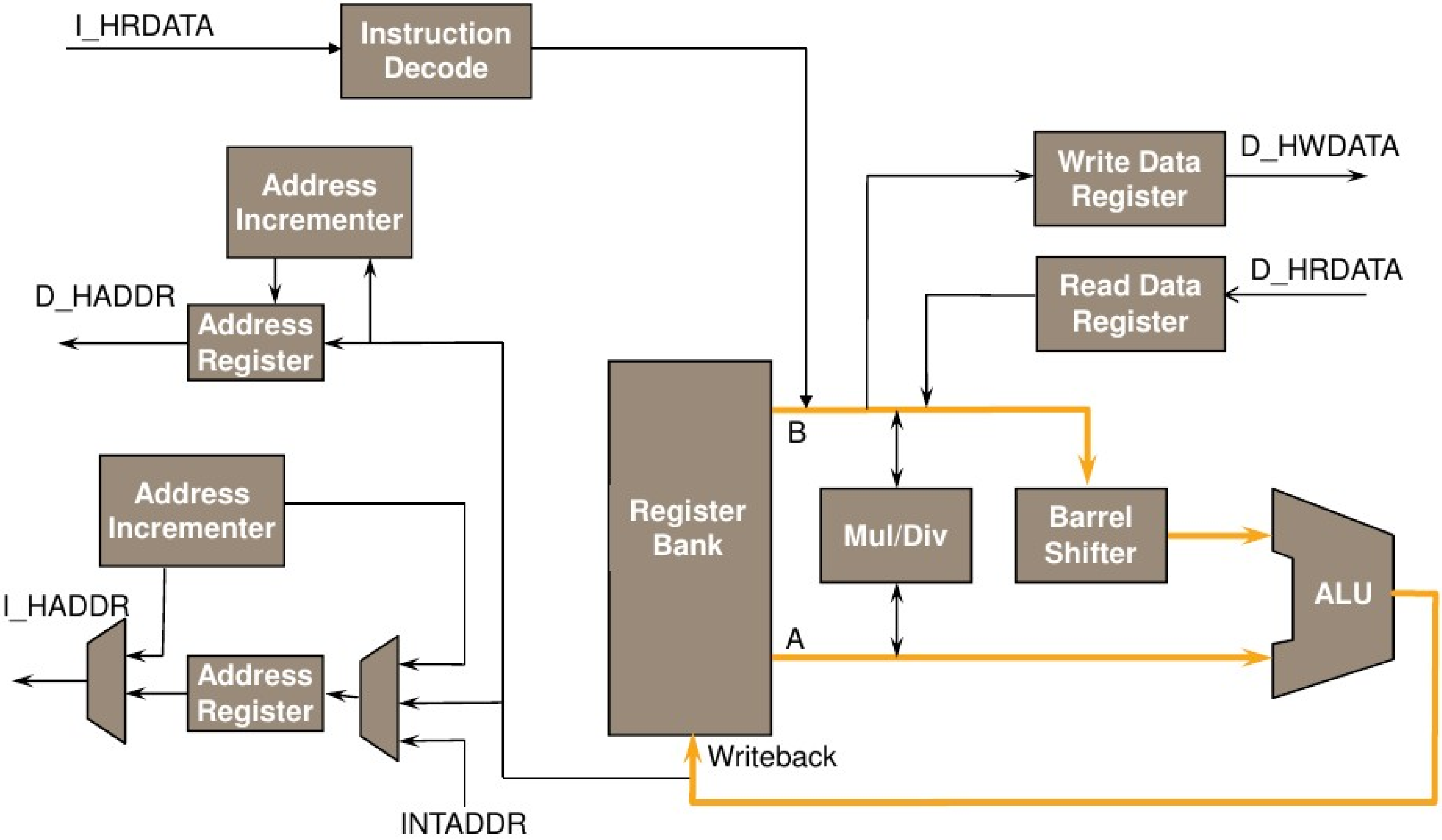
- Harvard can't use self-modifying code
- Harvard can fetch data and instructions at the same time
- Harvard necessary if program and data memories must use different technologies
  - Flash for program
  - SRAM for data

Processor core

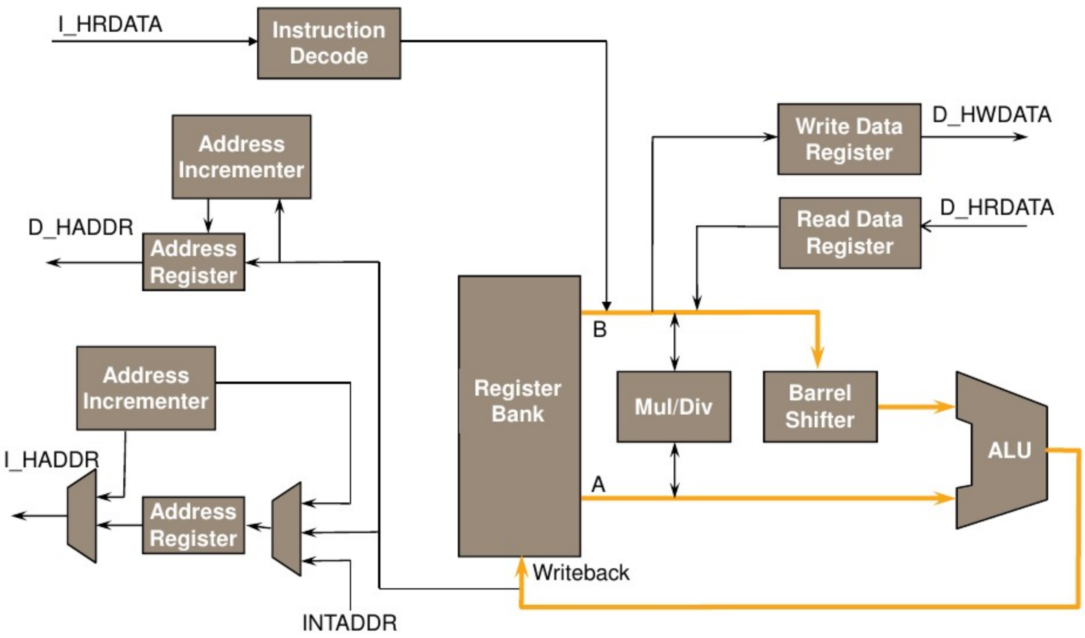
- Hardware unit in a computer designed to execute sequential programs
- Programs are sequences of simple instructions
- Loop:
  - Read next instruction from memory
  - Decode instruction
  - Execute instruction
    - Possibly write back result to memory

```
e59f0024
e5903000
e3530000
0a000003
e59f3014
0a000000
e12fff33
e8bd4008
eaffffda
00010f14
00000000
e28db000
e24dd00c
e50b0008
e50b100c
e51b2008
e51b3008
e0823003
e1a00003
e28bd000
e8bd0800
e12fff1e
```

ARM Cortex M3 Datapath



ARM Cortex M3 Datapath



Is this Harvard or von Neumann?

Instruction Set Architecture (ISA)

- Two main philosophies
- Complex Instruction Set Computer (CISC)
  - Many addressing modes
  - Many operations
- Reduced Instruction Set Computer (RISC)
  - Few and simple instruction formats
  - Instructions designed to be easy to pipeline

Instruction set characteristics

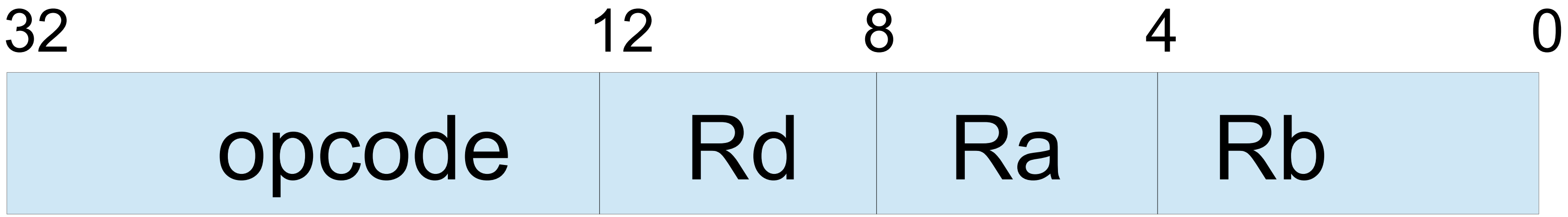
```
ADC $01f0
ADD R2, R3
ADD R3, R2, R3
```

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

RISC: Instruction word

- Fixed length
- A few different formats

```
ADD R3, R2, R3
e0823003
```



32 12 8 4 0

opcode Rd Ra Rb

RISC: Load-store architecture

- To simplify processor design
- Only special *load* and *store* instructions can access data memory
- Must therefore first load data from memory to registers before we can process the data

Implementations

- Successful CPU architectures have many implementations
  - Different capabilities, additional instructions (FPU, SIMD, ...)
  - Different performance (cache size, bus width, ...)
- Microcontroller variants
  - Different I/O units
  - On-chip memory
- Embedded designers have lots to choose from
- Biggest difference for SW is performance and I/O

Lecture overview

- Introduction to processors and instruction sets
- Assembly language
- ARM architecture and instructions
- From C to assembly

Assembly language

- Instructions in memory are binary

  `E0823003`

- Assembly language is a text file representation

  `ADD R3, R2, R3`

  - One-to-one with instructions (mostly)
  - Mnemonics + operands
  - Much easier to write for humans
    - No need to remember instruction formats and manually code instructions
  - Text file must be translated (“assembled”) to binary form before a processor can run it
    - Performed by the *assembler*

Symbols

- “Symbols” can be used instead of numbers and addresses
- Labels represent program address
- Constants represent other numbers

```
CONSTANT = 5

        LDR r0, [r8]   ; load from address in r8 to r0
label:
        ADD r0, r0, #CONSTANT
        B label
```

Directives and pseudo-ops

- Performed by assembler at assembly time
- Typically expands into a number of real instructions
  - Include other source files
  - Short hand notation of common instruction sequences
  - Allocate data areas

```
.include “file”
.word 5
```

Assembly summary

- Very processor specific
  - The CPU architecture document is your friend
- Even somewhat specific to the assembler
  - Pseudo-ops
  - labels and constants
  - comments
  - operand notation
- Principles are the same

Lecture overview

- Introduction to processors and instruction sets
- Assembly language
- ARM architecture and instructions
- From C to assembly

ARM processor

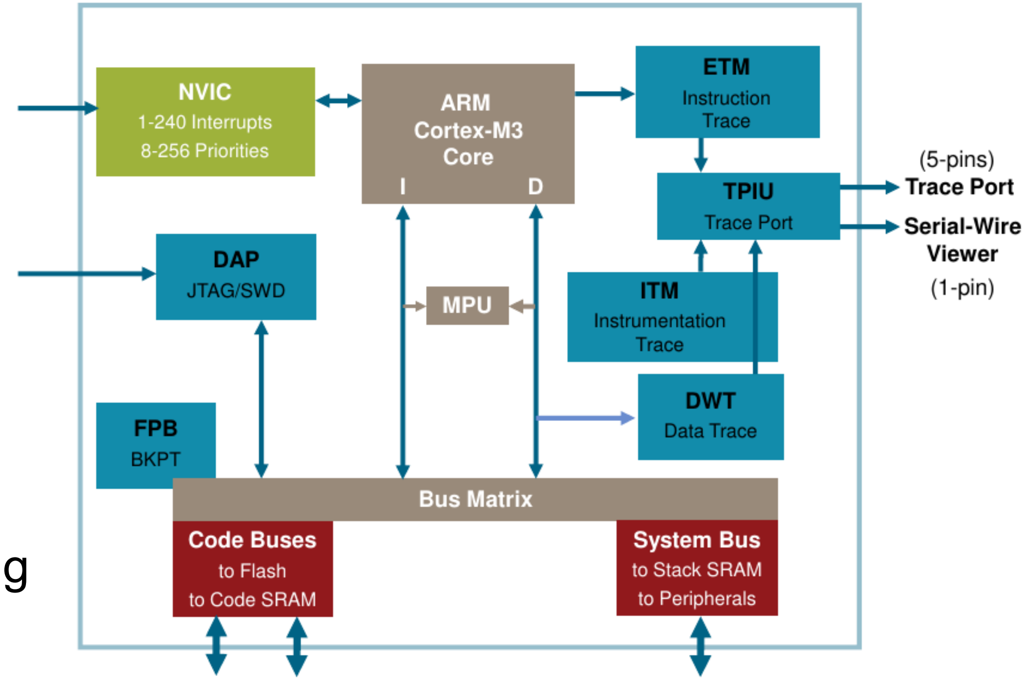
- ARM: Advanced RISC Machine
- The most widely used 32 bit processor architecture
  - Cell phones, smart phones, tablets
  - Lots of embedded systems
- ARM design started by Acorn Computer in 1983
  - Used to be Acorn RISC Machine
- ARM has HQ in Cambridge, UK
- Office in Trondheim
  - Media processor division (Mali GPU)

ARM architecture

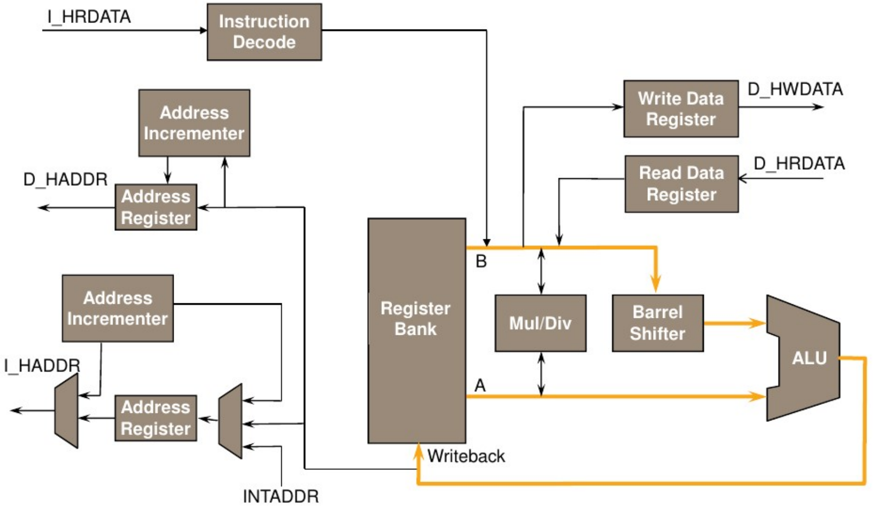
- RISC processor
  - Load-store architecture
  - Fixed length instructions
  - 3 operands
  - “Simple” hardware
- Instruction set
  - A few CISC features to improve code density
- Several architecture variants
  - Two main instruction set variants: 32bit and thumb

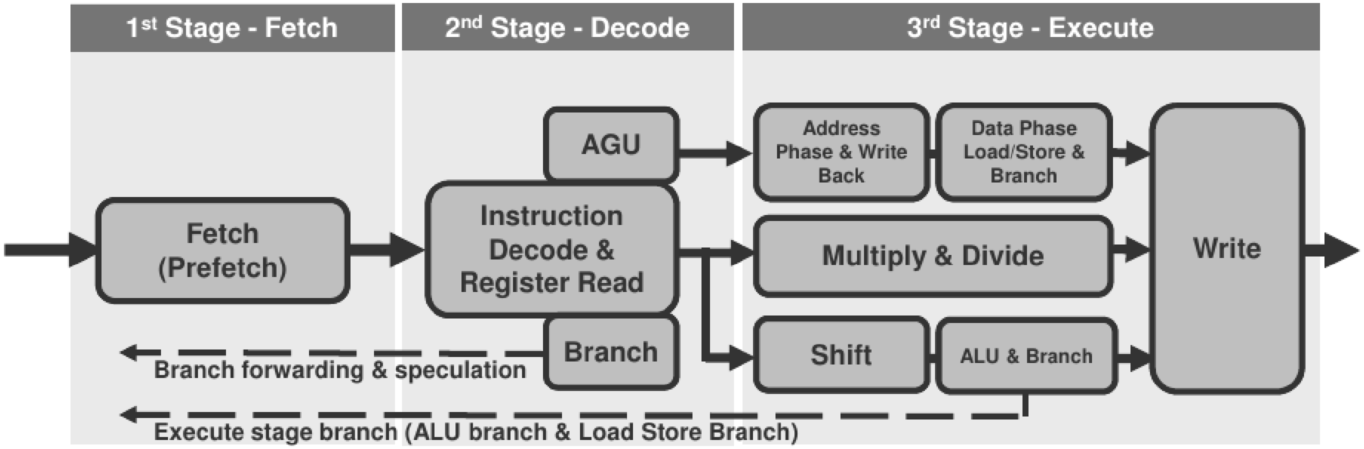
ARM architecture

- 32 bit core
- Bus interface
  - Unified bus interface
  - Both instruction and data
- Debugging support
  - Embedded HW support for doing single stepping, tracing etc.

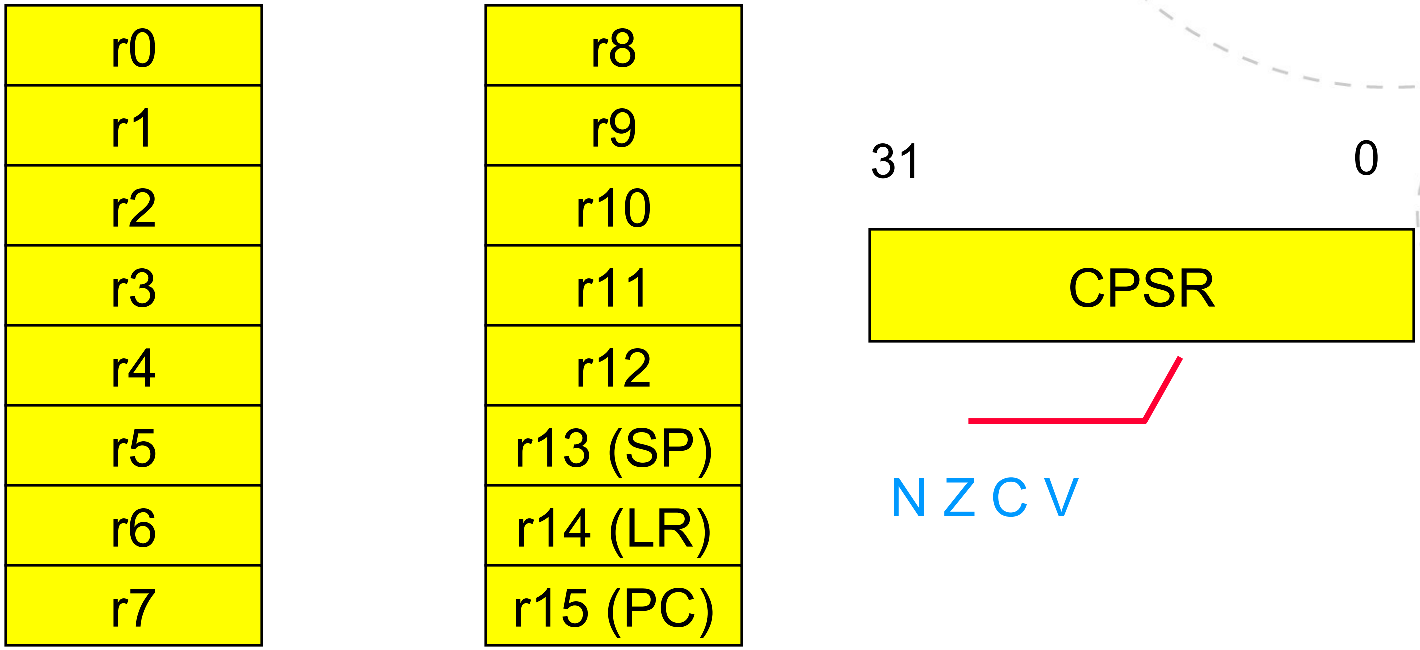


ARM processor organization




1st Stage - Fetch
2nd Stage - Decode
3rd Stage - Execute
Fetch (Prefetch)
AGU
Instruction Decode & Register Read
Branch
Branch forwarding & speculation
Address Phase & Write Back
Data Phase Load/Store & Branch
Multiply & Divide
Shift
ALU & Branch
Write
Execute stage branch (ALU branch & Load Store Branch)

ARM visible registers



r0

r1

r2

r3

r4

r5

r6

r7

r8

r9

r10

r11

r12

r13 (SP)

r14 (LR)

r15 (PC)

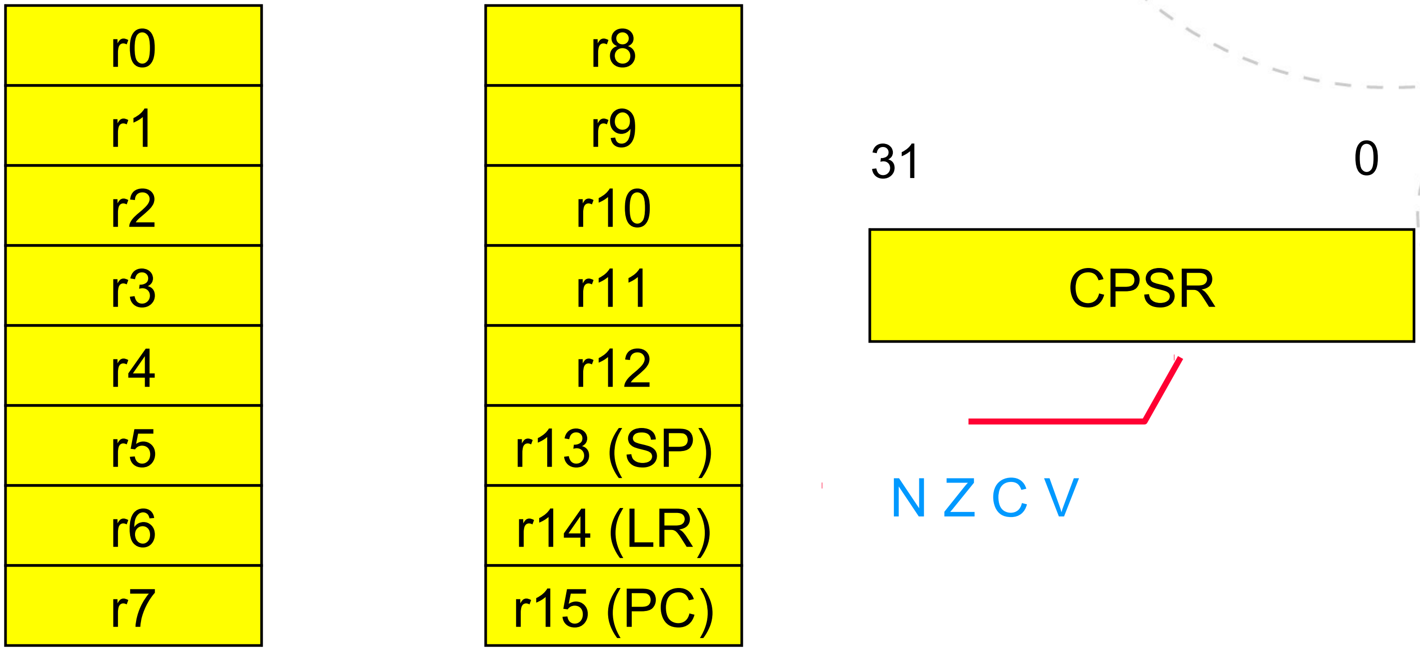
31

0

CPSR

N Z C V

ARM visible registers



r0
r1
r2
r3
r4
r5
r6
r7

r8
r9
r10
r11
r12
r13 (SP)
r14 (LR)
r15 (PC)

31
0
CPSR
N Z C V

Why have PC, LR and SP in the regfile?

ARM data instructions

- ADD, ADC
- SUB, SBC
- MUL, MLA
- AND, ORR, EOR
- LSL, LSR, ASL, ASR
- ROR, RRX

- MOV R0, R1

Data instruction varieties

- With or without carry
  - ADD R1, R2, R3 ; R1 = R2 + R3
  - ADC R1, R2, R3 ; R1 = R2 + R3 + C
  - What is the point of adding carry?
- Logical or arithmetic shift
  - LSR: shift and insert 0
  - ASR: shift and insert sign bit
- Rotate and extend with carry
  - 33-bit rotate

ARM data instruction addressing modes

ARM is a load-store machine. Which addressing modes does data instructions have?

ARM data instruction addressing modes

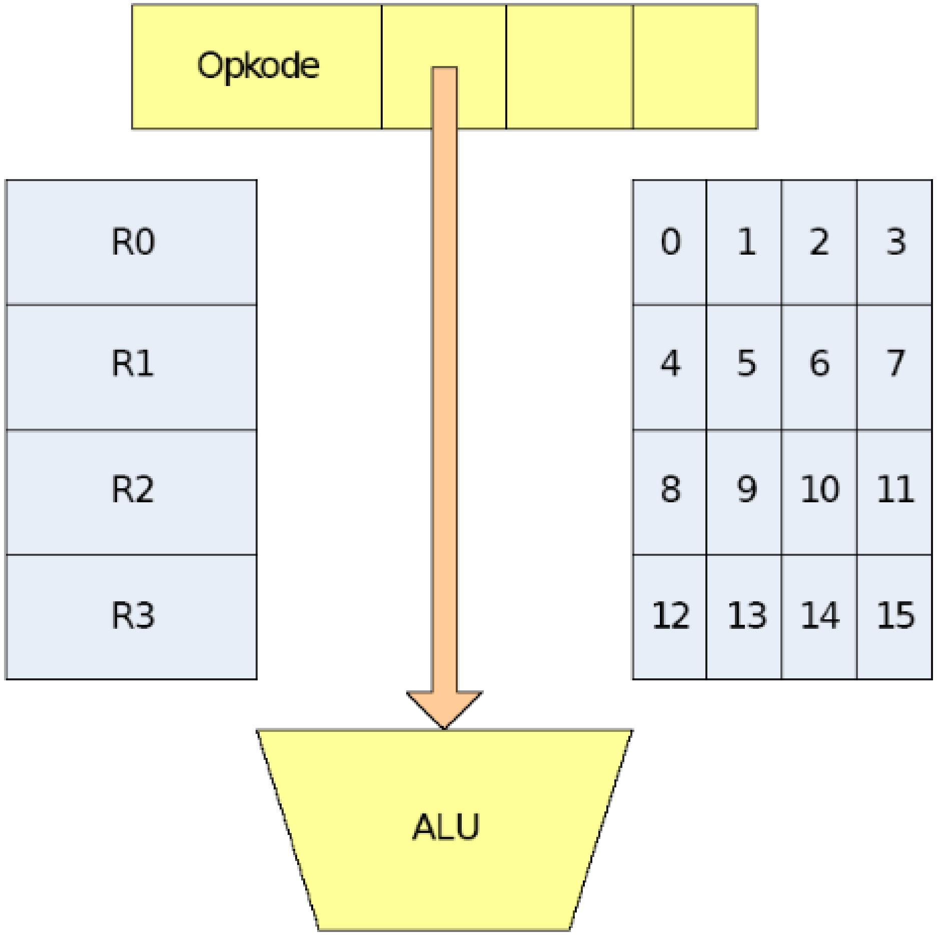
- Immediate

  ADD R1, R2, #5
- Register

  ADD R1, R2, R3

Immediate addressing

- Immediate
  ADD R1, R2, #5



Immediate addressing

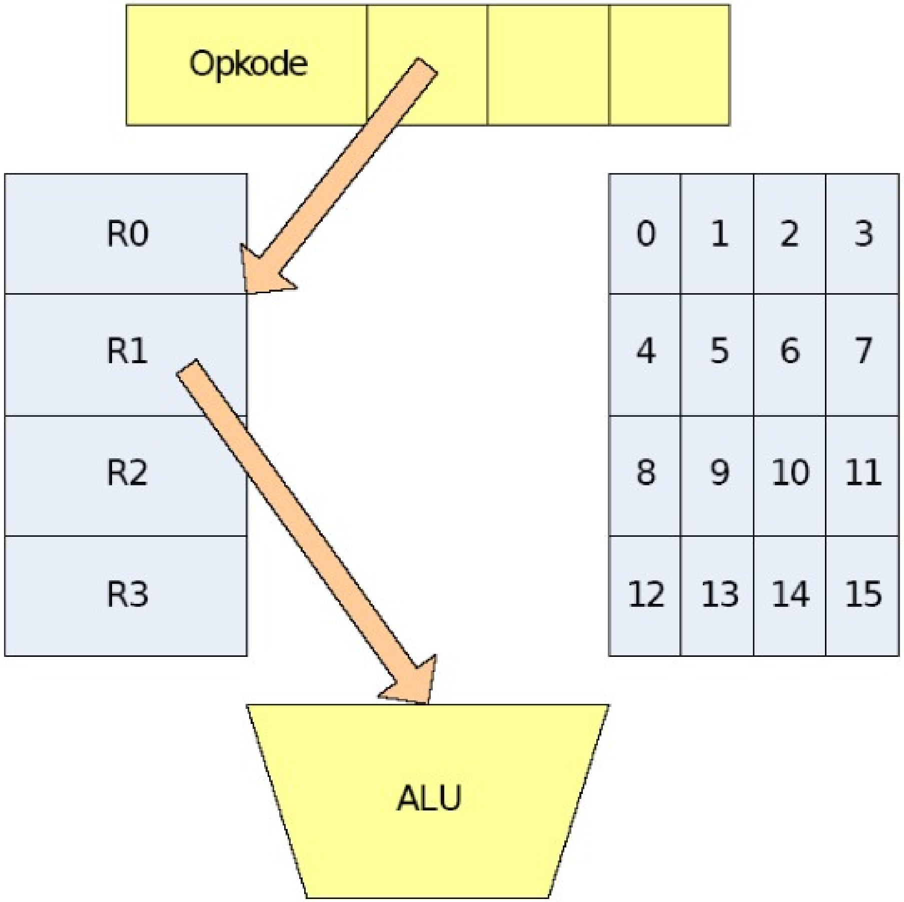
- Size restriction
  - Registers are 32 bit
  - Instructions are 32 bit
    - Only 8 bits for immediate field
- How can we load 32 bit numbers into registers?

Immediate addressing

- Size restriction
  - Registers are 32 bit
  - Instructions are 32 bit
    - Only 8 bits for immediate field
- How can we load 32 bit numbers into registers?
  - Store constant in memory and load to register with LDR
  - Use MOV with 4 bit rotate to load 8 bit value to any location

Register addressing

- Register addressing
  ADD R3, R2, R1

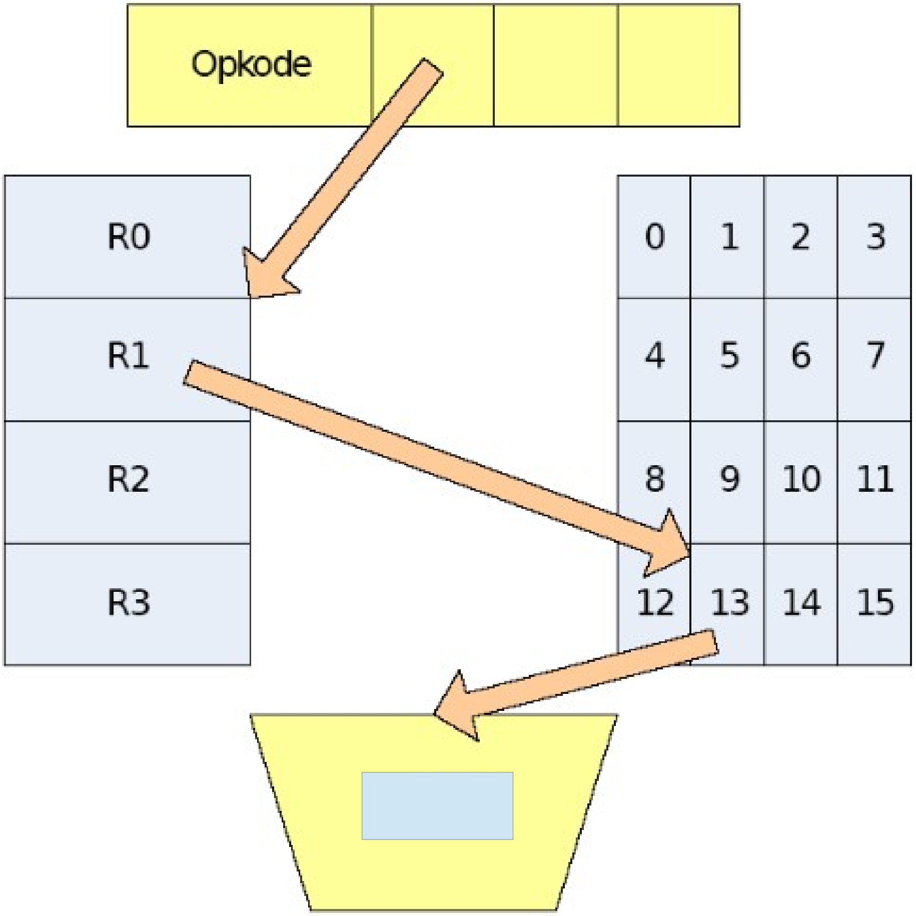


ARM load-store instructions

- LDR, LDRH, LDRB
  - word, half-word, byte
- STR, STRH, STRB

ARM load-store instruction addressing modes

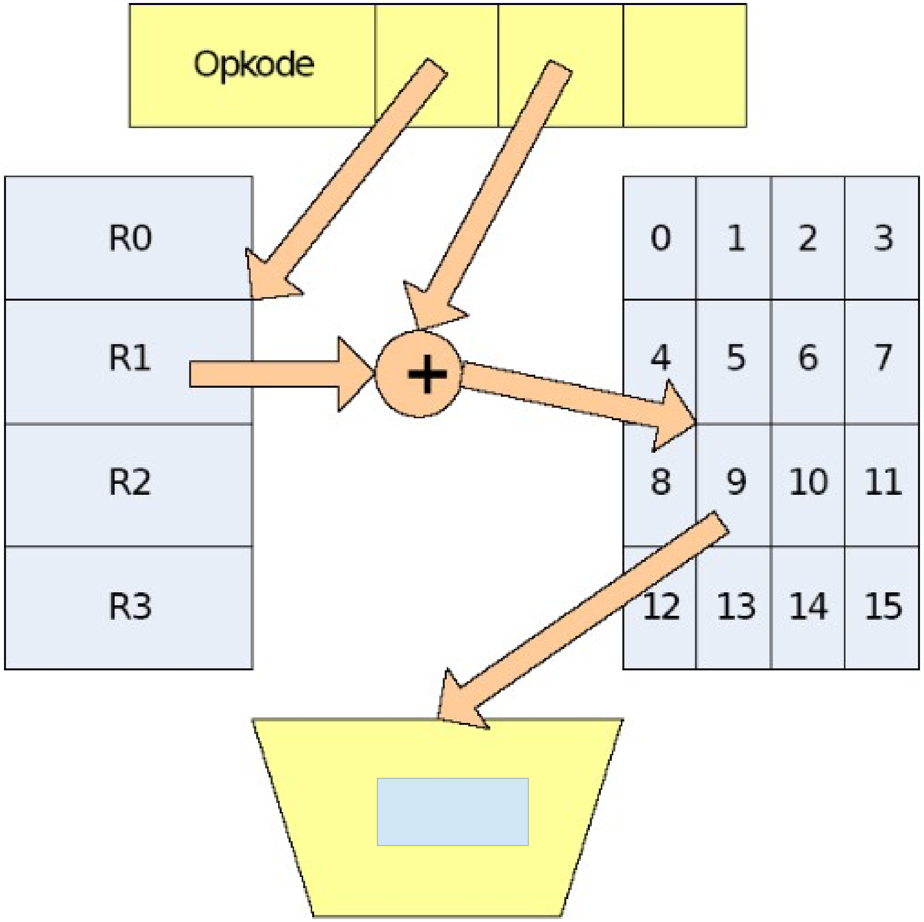
- Register indirect
  LDR R0, [R1]



ARM load-store instruction addressing modes

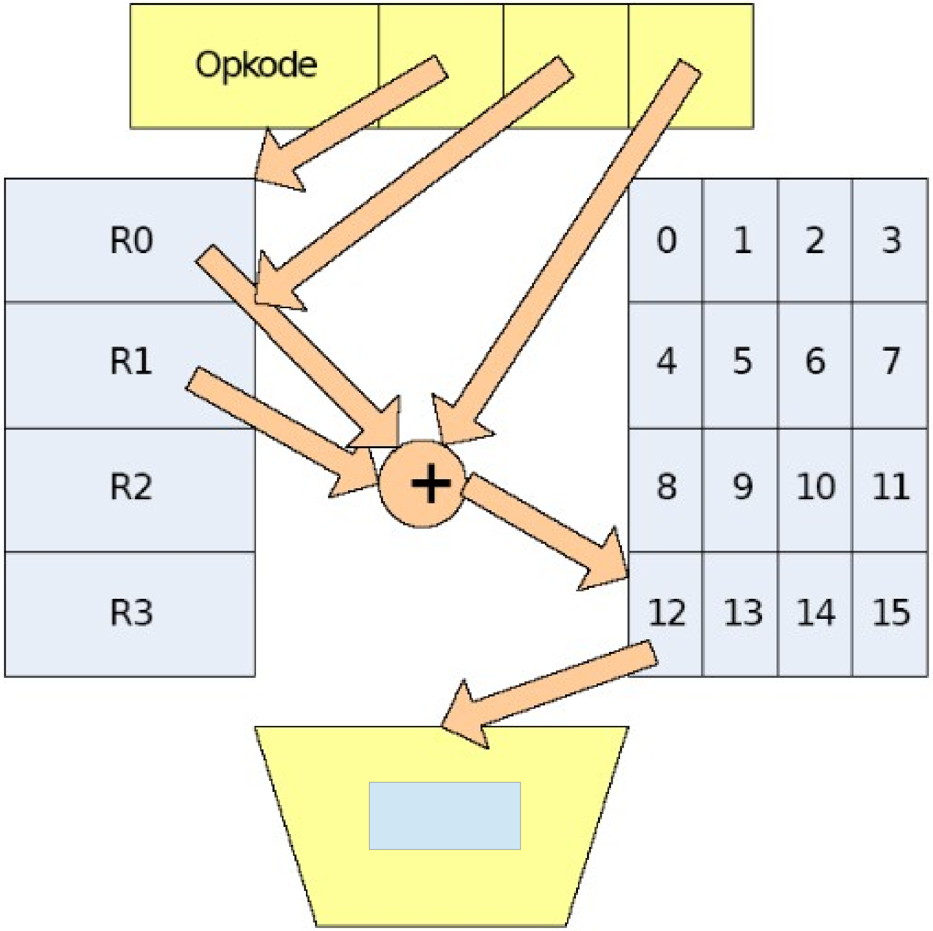
- Indexed

  LDR R0, [R1, #4]



ARM load-store instruction addressing modes

- Base indexed
  LDR R2, [R0, -R1]



ARM flow control

```
B #100 ; jump 100 words forward
```

- PC relative
  - PC = PC + imm * 4
  - Most branches are short
- Assembler calculates the offset automatically

```
label B label
```

- How to do a long jump?

Conditional instructions

- Comparison instructions
  - CMP
  - CMN
  - TST
  - TEQ
  - Sets NZCV bits of CPSR
- BEQ, BNE, BCS, BCC, BMI, BPL, BVS, BVC, BHI, BLS, BGE, BLT, BGT, BLE
- Also MVEQ, ADDEQ, etc.
  - All instructions can have condition code

Lecture overview

- Introduction to processors and instruction sets
- Assembly language
- ARM architecture and instructions
- From C to assembly

Example: C to ARM assembly

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

x = a + b;
```

- Useful instructions:
  - LDR: load from mem
  - ADD
  - STR: store to mem
- Useful addressing modes:
  - PC relative: LDR Rx, rel_address
  - Register: ADD Rd, Ra, Rb
- Useful pseudo-ops:
  - Reserve dataword: .word X

Example: C to ARM assembly

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

x = a + b;
```

```
        ldr     r2, vars
        ldr     r3, vars+4
        add     r2, r2, r3
        str     r2, vars+8

        ...

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

- Useful instructions:
  - LDR: load from mem
  - ADD
  - STR: store to mem
- Useful addressing modes:
  - PC relative: LDR Rx, rel_address
  - Register: ADD Rd, Ra, Rb
- Useful pseudo-ops:
  - Reserve dataword: .word X

Example: C to ARM assembly

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

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

- Use conditional instructions (CMP, xxxEQ, xxxNE)
- LDR, STR, B
- Branch: B label

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
         bge      else
         mov      r2, #1
         b        endif
else:
         mov      r2, #2
endif:
         str      r2, vars+8
         ...

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

- Use conditional instructions (CMP, xxxGE, xxxLT)
- LDR, STR, B
- Branch: B label

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
```

- Use conditional instructions (CMP, xxxGE, xxxLT)
- LDR, STR, B
- Branch: B label

ARM subroutines

- Branch-and-link instruction

```
BL label
```

  - Copy PC (R15) to LR (R14)
  - Jump to `label`
- Function arguments
  - Registers or stack
- Return from subroutine

```
MOV PC, LR
```

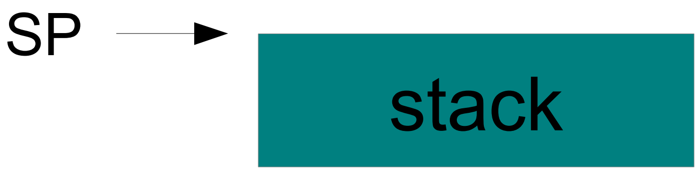
ARM subroutines: Call stack

- Must avoid destroying registers when executing subroutines
  - Push registers on the stack
  - Execute subroutine code
  - Pop registers from the stack
- Special instructions to push and pop multiple registers
  - STMFD SP!, {R4, R5, R6}
  - LDMFD SP!, {R4, R5, R6}
- Local subroutine variables can be allocated on the stack
  - Save pointer to stack frame (R11 is the frame pointer)
  - Advance stack to make room for variables
  - Variables are referenced relative to frame pointer

Stack frames

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

```
test(2);
```



SP

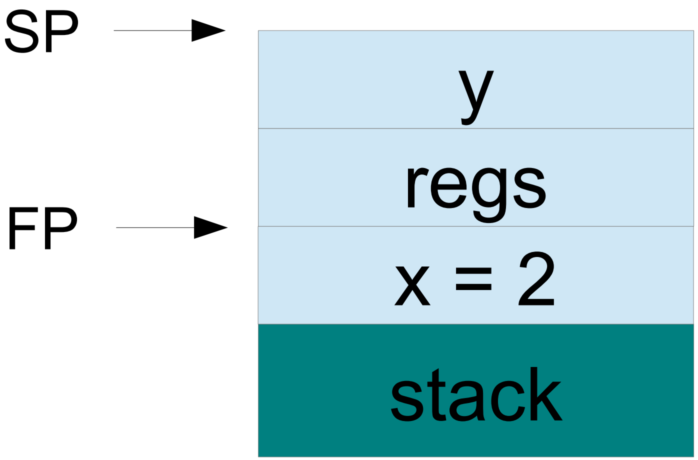
stack

Stack frames

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

---

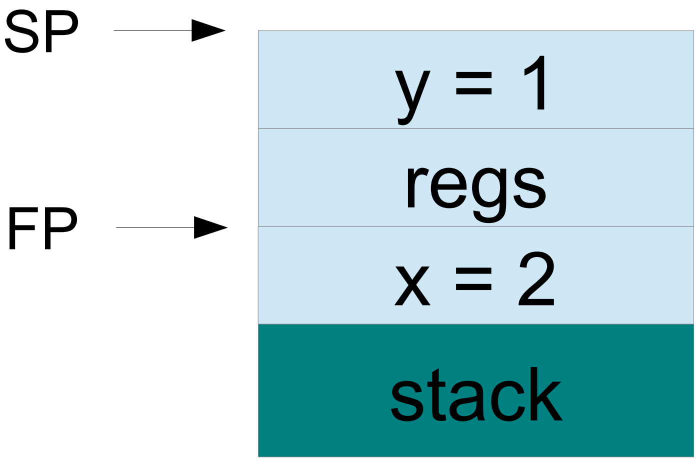
test(2);



Stack frames

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

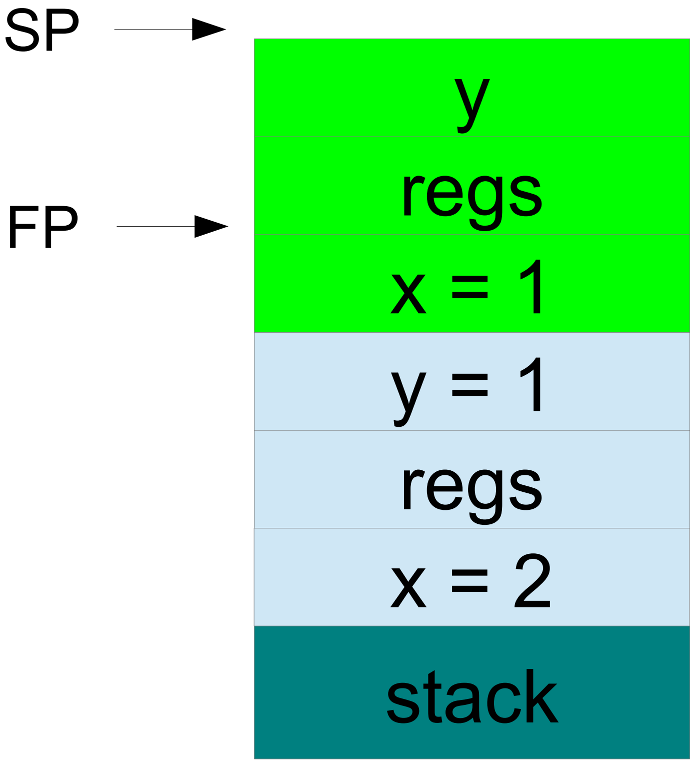
```
test(2);
```



Stack frames

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

```
test(2);
```



SP

FP

y

regs

x = 1

y = 1

regs

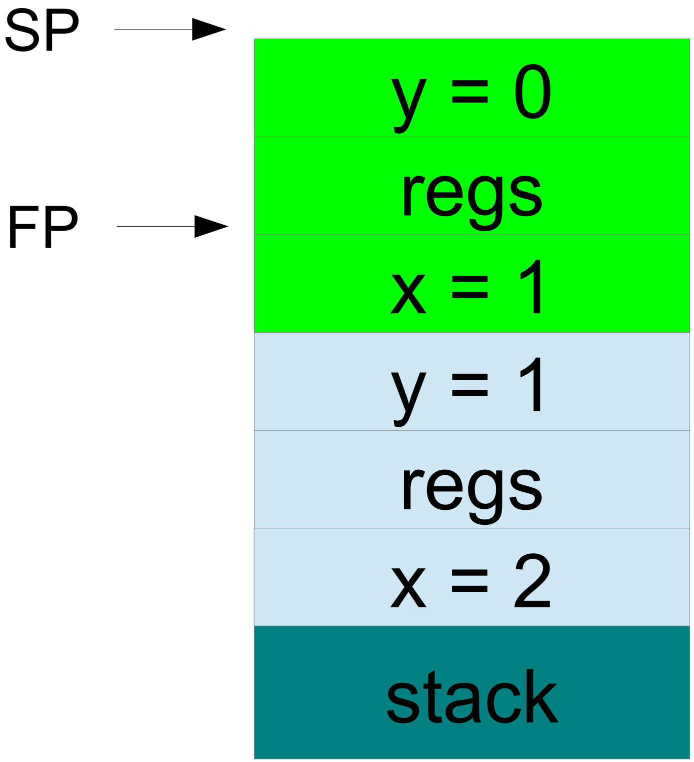
x = 2

stack

Stack frames

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

```
test(2);
```



SP

FP

y = 0

regs

x = 1

y = 1

regs

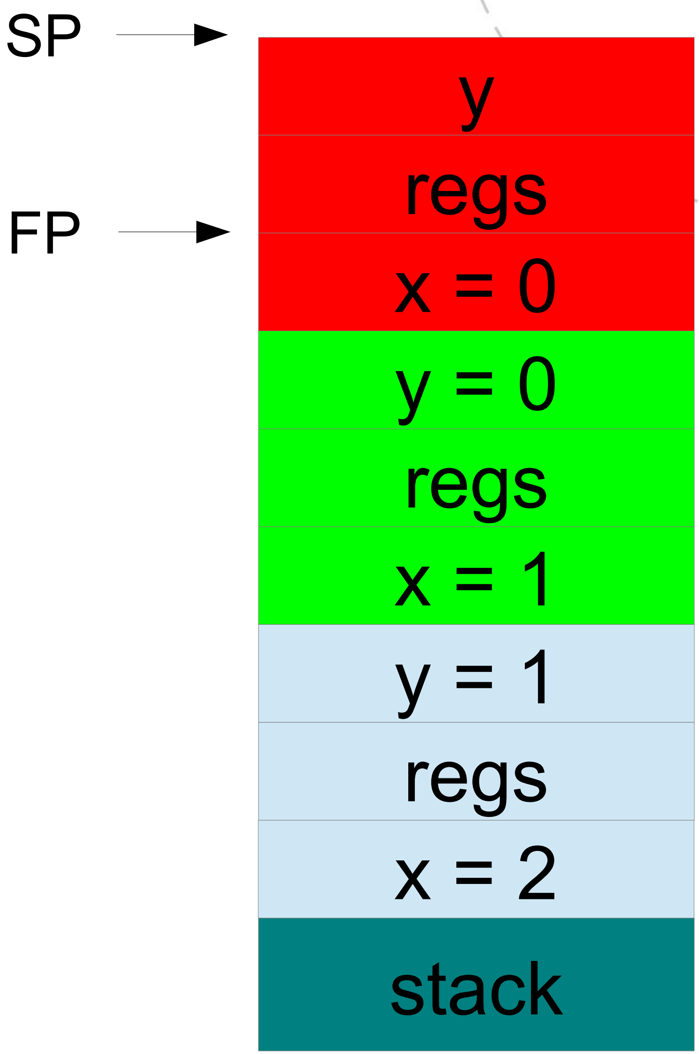
x = 2

stack

Stack frames

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

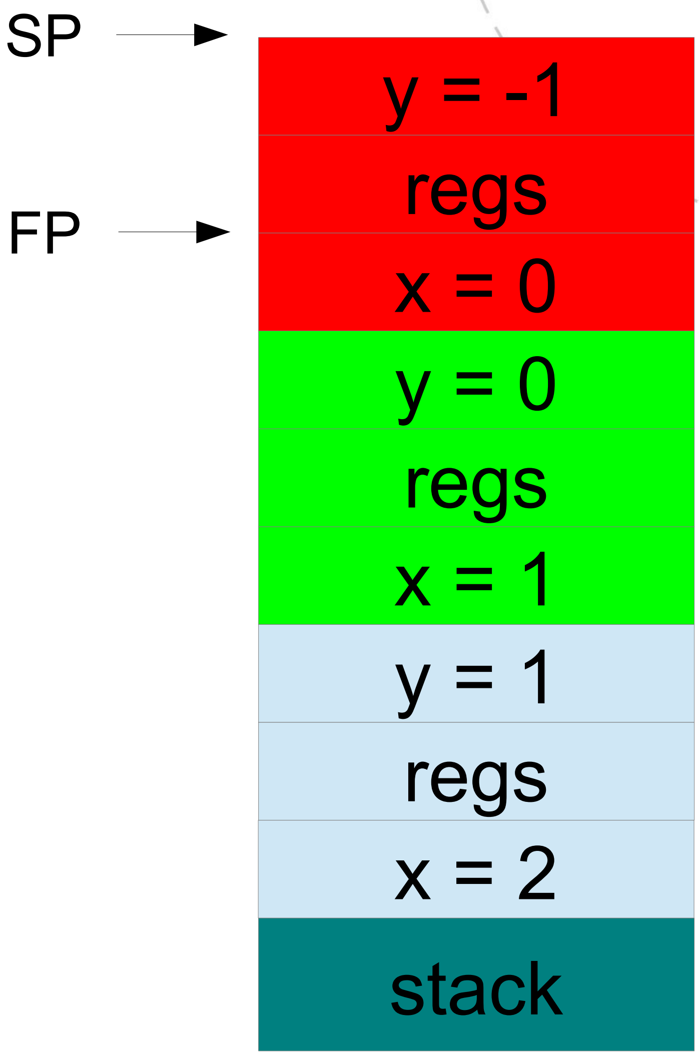
```
test(2);
```



Stack frames

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

```
test(2);
```



SP

FP

y = -1

regs

x = 0

y = 0

regs

x = 1

y = 1

regs

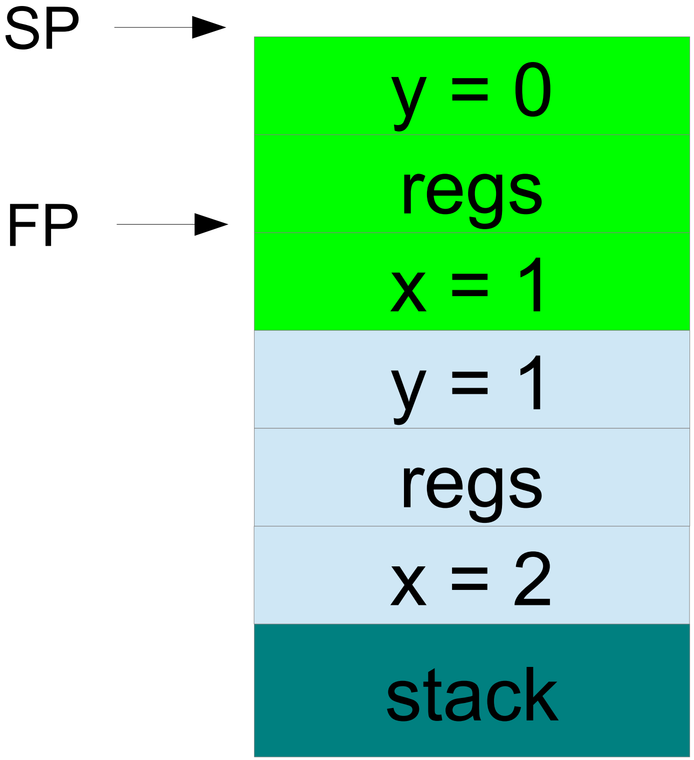
x = 2

stack

Stack frames

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

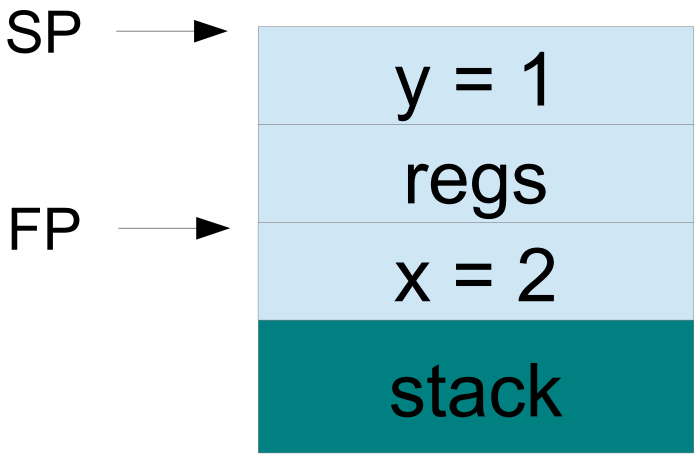
```
test(2);
```



Stack frames

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

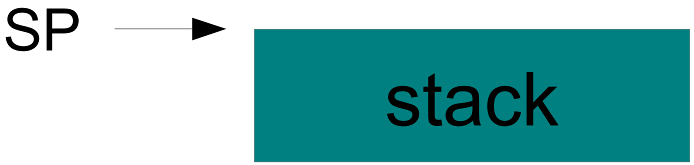
```
test(2);
```



Stack frames

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

```
test(2);
```



ARM register allocation

r0 (arg)	r8
r1 (arg)	r9
r2 (arg)	r10
r3 (arg)	r11 (FP)
r4	r12
r5	r13 (SP)
r6	r14 (LR)
r7	r15 (PC)

ARM subroutine example

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

void main(void) {
  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}
```

Next lectures

- Tomorrow 12:15 (R40): Exercise introduction
- Next thursday: Guest lecture, Energy Micro