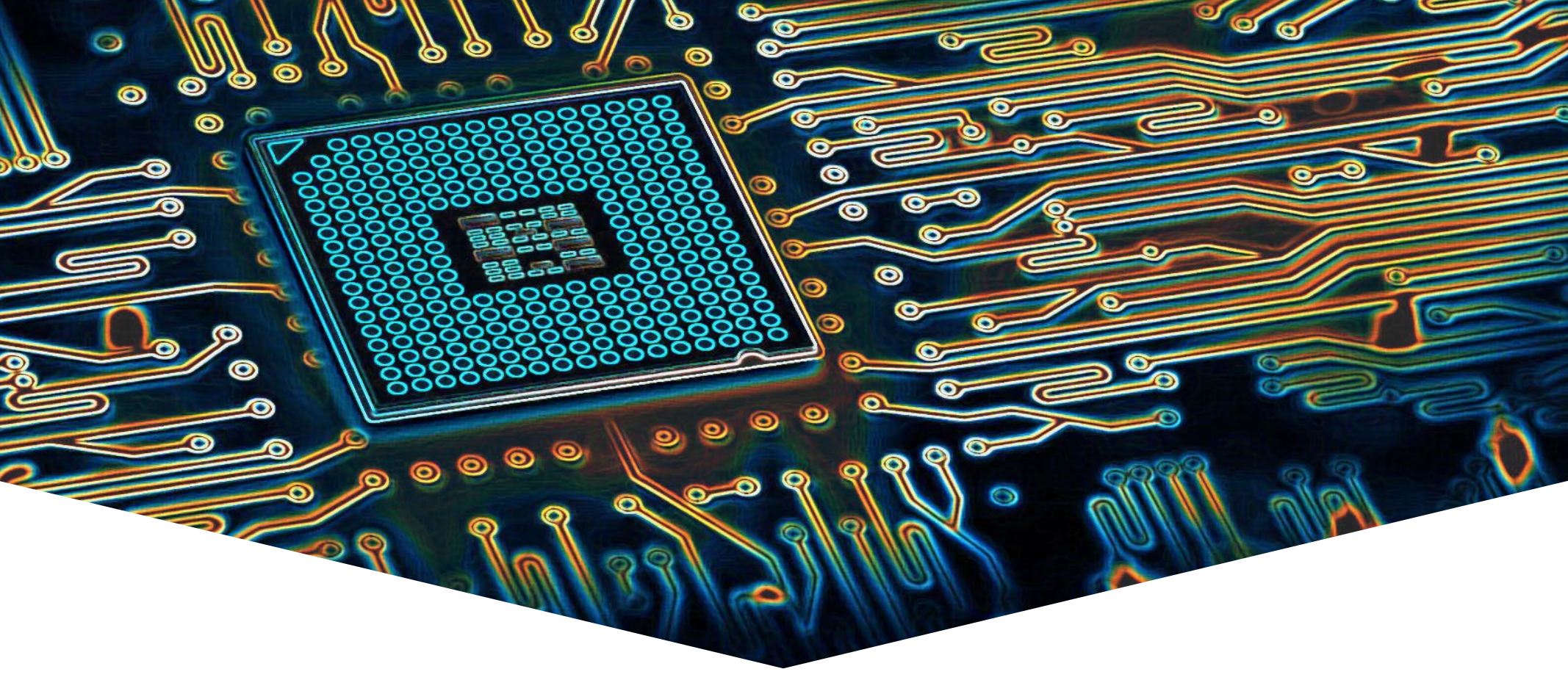


RTS10 Week 1

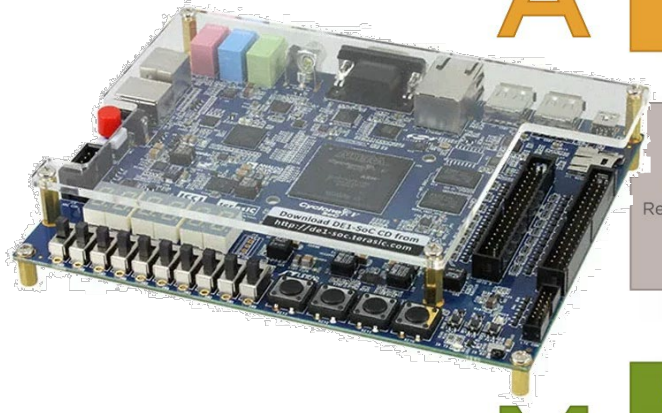
Leerdoelen Week 1 Theoriedeel

Leerdoelen week 1 theoriedeel. Je leert:

- de ARM Cortex processor familie kennen;
- een vereenvoudigde versie van de ARMv7-M architectuur, de LEGv7-M architectuur, kennen;
- een vereenvoudigde versie van de Thumb instructieset, de Pinky instructieset, kennen;
- assembleertaalprogramma's te schrijven in Pinky;
- Pinky assembly om te zetten naar machinecode (encoden);
- Pinky machinecode om te zetten naar assembly (decoden).

Studiemateriaal: De LEGv7 architectuur en de Pinky instructieset

ARM Cortex Processor Family



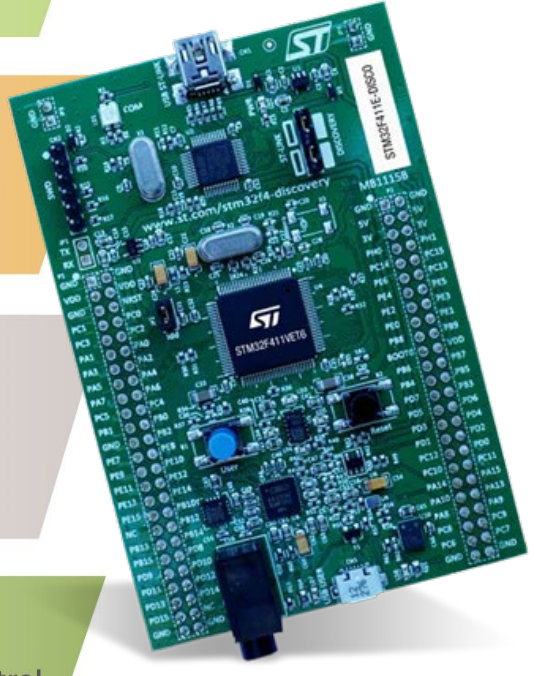
A

Cortex-A9	**Cortex-A15** ARMv7-A High Performance 32-bit CPU with enterprise class feature set	**Cortex-A17** ARMv7-A High Performance 32-bit CPU with lower power and smaller area	**Cortex-A57** ARMv8-A Highest performance 64/32-bit CPU	High Performance
	Cortex-A5 ARMv7-A Smallest and lowest power CPU	**Cortex-A7** ARMv7-A High efficiency 32-bit CPU big LITTLETM compatible	**Cortex-A53** ARMv8-A High efficiency 64/32-bit CPU big LITTLETM compatible	High Efficiency
	Cortex-R4 Real-time standard	**Cortex-R5** Functional safety	**Cortex-R7** High performance 4G modern and storage	Real-time

Enhanced system integration features

M

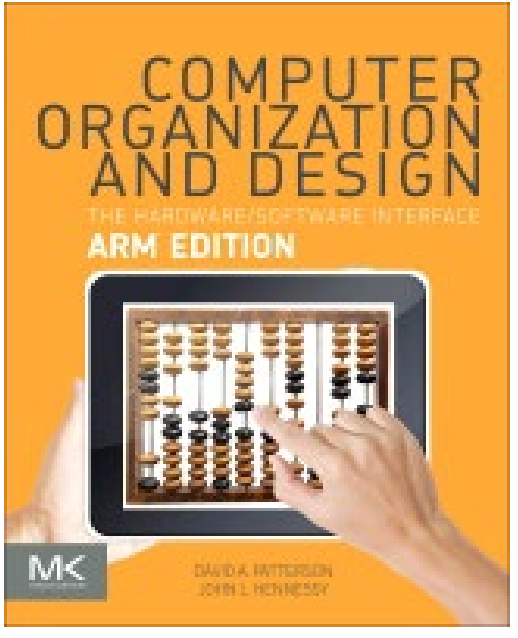
Cortex-M0 Lowest cost Lowest power **Cortex-M0+** Highest energy efficiency	**Cortex-M3** Performance efficiency	**Cortex-M4** Mainstream Control & DSP	**Cortex-M7** Maximum Performance Control & DSP	Control



Source Reference

The material in this presentation is inspired by chapter 2 of the book:

- Computer Organization and Design
  - The Hardware / Software Interface
    - ARM Edition
    - David A. Patterson, John L. Hennessy
    - ISBN: 9780128017333



Main differences:

- The book uses ARMv8 Cortex-A , I use ARMv7 Cortex-M
- The book uses ARM instructions, I use Thumb instructions

Agenda

- Instruction Set
- Operations and Operands
- Representing Instructions in the Computer
- Logical Operations
- Instructions for Making Decisions
- LEGv7 Addressing for 32-bits Literals
- Supporting Functions in Computer Hardware

Agenda

- **Instruction Set**
- Operations and Operands
- Representing Instructions in the Computer
- Logical Operations
- Instructions for Making Decisions
- LEGv7 Addressing for 32-bits Literals
- Supporting Functions in Computer Hardware

Instructions: The language of the computer

- How to tell a computer what to do?

- Bits are the letters of the computer
- Instructions are the words of the computer
  - An instruction is a collection of bits
- The instruction set is the vocabulary of the computer
  - An instruction set is a collection of instructions

Instruction Set

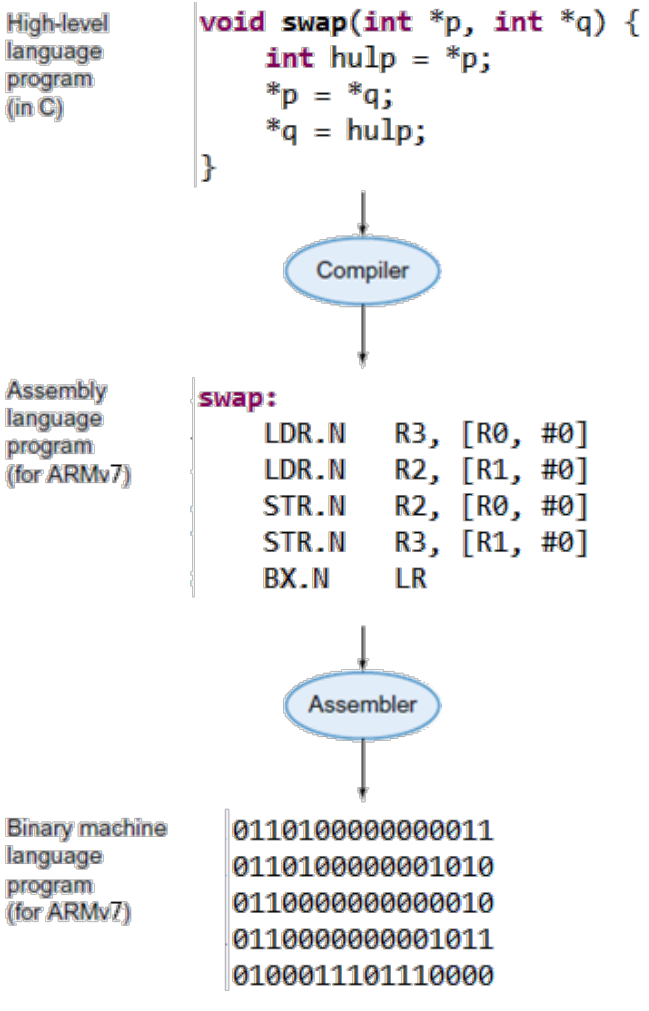
- The repertoire of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common

The ARMv7 Thumb Instruction Set

- A subset, called LEGv8 Pinky, used as the example throughout these lessons
- Commercialized by ARM Holdings (www.arm.com)
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Typical of many modern ISAs
  - See De LEGv7 architectuur en de Pinky instructieset

Levels of Program Code

- High-level language
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability
- Assembly language
  - Textual representation of instructions
- Hardware representation
  - Binary digits (bits)
  - Encoded instructions and data



Agenda

- Instruction Set
- **Operations and Operands**
- Representing Instructions in the Computer
- Logical Operations
- Instructions for Making Decisions
- Supporting Functions in Computer Hardware
- LEGv7 Addressing for 32-bits Literals

Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination

```
ADDS.N a, b, c  // a gets b + c
SUBS.N a, b, c  // a gets b - c
```

- S means that the instruction is updating condition flags (N = Negative, Z = Zero, V = oVerflow, C = Carry)
- N means Narrow (16-bits instruction)

Arithmetic Example

- C code:

```
f = (g + h) - (i - h);
```

- Compiled LEGv7 Pinky code:

```
ADDS.N  t0, g, h
SUBS.N  t1, i, h
SUBS.N  f, t0, t1
```

Processor does not work with variables but with registers

Memory Hierarchy

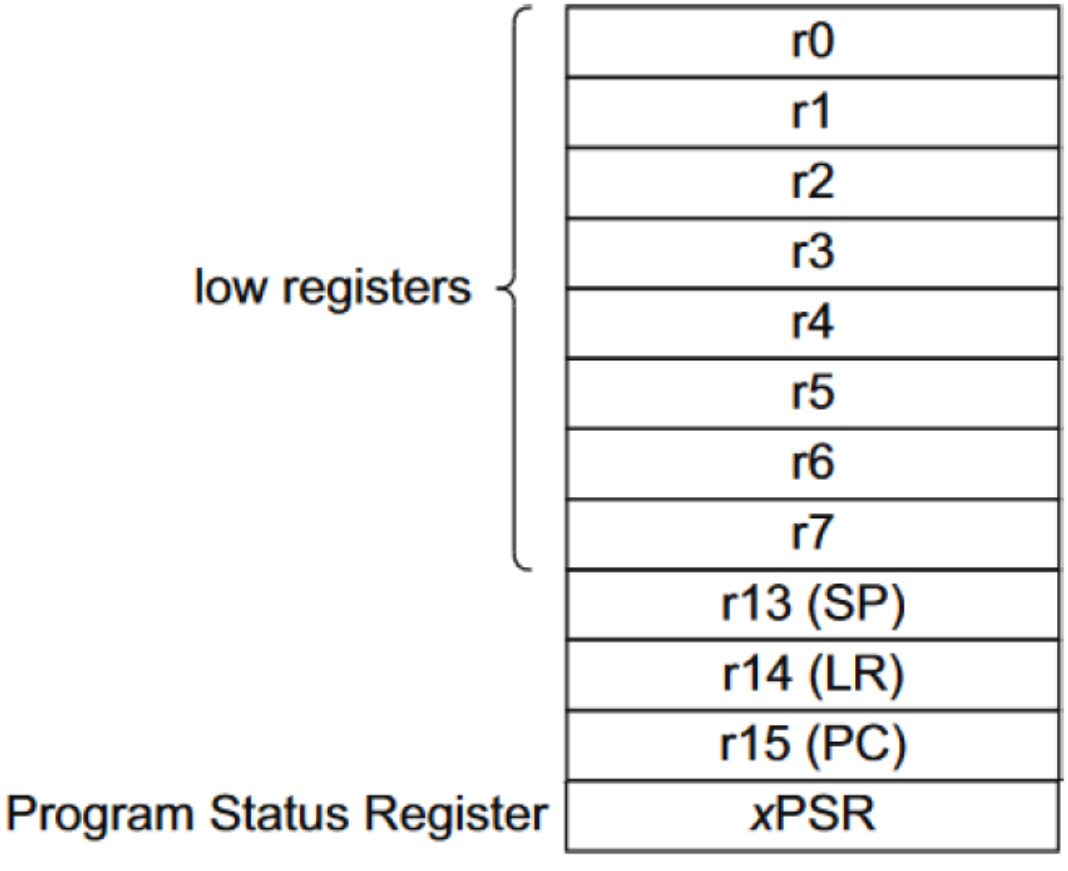
- The ALU can only operate on values that are inside the registers
- Not directly from the main memory!
- Need to LOAD stuff from main memory

Register Operands

- Arithmetic instructions use register operands

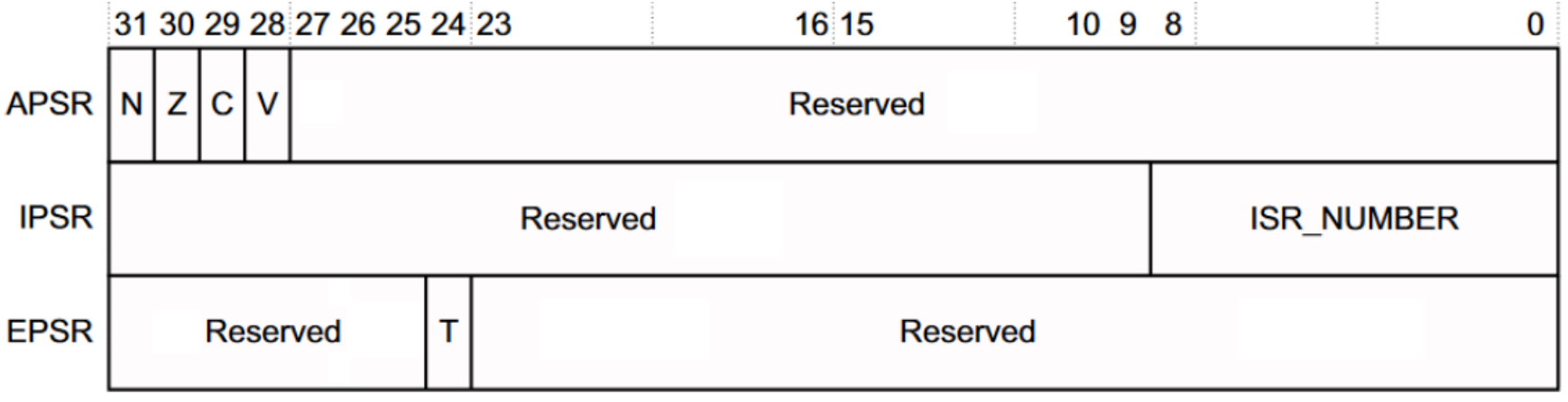
- LEGv7 has an 8 × 32-bits register file
  - Use for frequently accessed data
  - 32-bits data is called a “word”
    - 8 x 32-bits general purpose registers R0 to R7

LEGv8 Registers



Program Status Register

- Consists of three overlapping registers
  - Application Program Status Register (APSR)
  - Interrupt Program Status Register (IPSR)
  - Execution Program Status Register (EPSR)



T bit indicates that Thumb (Pinky) instructions are executed

Arithmetic Example

- C code:

```
f = (g + h) - (i - h);
```

  - f, g, h, i in R4, R5, R6, R7
- Compiled LEGv7 Pinky code:

```
ADDS.N  R0, R5, R6
SUBS.N  R1, R7, R6
SUBS.N  R4, R0, R1
```

Arithmetic Example (Alternative)

- C code:

```
f = (g + h) - (i - h);
```

  - f, g, h, i in R4, R5, R6, R7
- Compiled LEGv7 Pinky code:

```
ADDS.N   R4, R5, R6
SUBS.N   R4, R4, R7
ADDS.N   R4, R4, R6
```

No temporally registers needed

Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- LEGv7 does not require words to be aligned in memory, except for instructions (must be halfword aligned) and the stack (must be word aligned)
  - But unaligned loads/stores take 2 (halfword, not word aligned) or 3 (not halfword aligned) clock cycles extra

Load / Store instructions

Instructie	Betekenis	N	Z	C	V	Cycles
LDR.N <Rt>,=<label>	<Rt> ← <label>	-	-	-	-	2
LDR.N <Rt>, [<Rn>,#<imm7>]	<Rt> ← [<Rn> + <imm7>]	-	-	-	-	2
LDR.N <Rt>,[<Rn>,<Rm>]	<Rt> ← [<Rn> + <Rm>]	-	-	-	-	2
STR.N <Rt>, [<Rn>,#<imm7>]	<Rt> → [<Rn> + <imm7>]	-	-	-	-	1
STR.N <Rt>,[<Rn>,<Rm>]	<Rt> → [<Rn> + <Rm>]	-	-	-	-	2

`#<imm7>` is a 7-bits unsigned immediate offset (0 t/m 127) that is dividable by 4

Arithmetic Example (Alternative)

- C code:

```
int a[] = {1,2,3,4,5,6,7,8,9};
a[8] = h + a[3];
```

  - h in R4, base address of a in R5
- Compiled LEGv7 Pinky code:

```
LDR.N   R0, [R5, #12]
ADDS.N  R1, R4, R0
STR.N   R1, [R5, #32]
```

Registers versus Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!

Immediate Operands

- Constant data specified in an instruction
  - `<imm3>` = 0 t/m 7, `<imm8>` = 0 t/m 255

Instructie	Betekenis	N	Z	C	V	Cycles
ADDS.N <Rdn>,#<imm8>	<Rdn> ← <Rdn> + <imm8>	↕	↕	↕	↕	1
ADDS.N <Rd>,<Rn>,#<imm3>	<Rd> ← <Rn> + <imm3>	↕	↕	↕	↕	1
ADDS.N <Rd>,<Rn>,<Rm>	<Rd> ← <Rn> + <Rm>	↕	↕	↕	↕	1
SUBS.N <Rdn>,#<imm8>	<Rdn> ← <Rdn> - <imm8>	↕	↕	↕	↕	1
SUBS.N <Rd>,<Rn>,#<imm3>	<Rd> ← <Rn> - <imm3>	↕	↕	↕	↕	1
SUBS.N <Rd>,<Rn>,<Rm>	<Rd> ← <Rn> - <Rm>	↕	↕	↕	↕	1

- Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

Signed and Unsigned Numbers

- **I** skip this part of the lecture because I assume you already now this stuff.
- **You** maybe have to study it yourself!

Unsigned Binary Integers

- Range: 0 to $+2^n - 1$

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits
  - 0 to +4,294,967,295

2s-Complement Signed Integers

- Range: -2^{n-1} to $+2^{n-1} - 1$

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 32 bits
  - –2,147,483,648 to +2,147,483,647

2s-Complement Signed Integers

>>

- Bit 31 is sign bit
  - `1` for negative numbers
  - `0` for non-negative numbers
- $-(-2^{n-1})$ can't be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
  - `0:  0000 0000 … 0000`
  - `–1:  1111 1111 … 1111`
  - Most-negative: `1000 0000 … 0000`
  - Most-positive: `0111 1111 … 1111`

Signed Negation

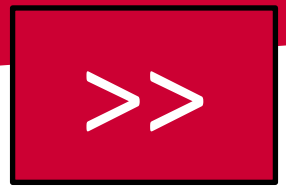
- Complement and add 1
  - Complement means 1 → 0, 0 → 1

$$x + \bar{x} = 1111...111_2 = -1$$

$$\bar{x} + 1 = -x$$

- Example: negate +2
  - +2 = 0000 0000 … 0010_{two}
  - −2 = 1111 1111 … 1101_{two} + 1
    = 1111 1111 … 1110_{two}

Sign Extension



- Representing a number using more bits
  - Preserve the numeric value

- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s

- Examples: 8-bit to 16-bit
  - `+2: 0000 0010 => 0000 0000 0000 0010`
  - `–2: 1111 1110 => 1111 1111 1111 1110`

Agenda

- Instruction Set
- Operations and Operands
- **Representing Instructions in the Computer**
- Logical Operations
- Instructions for Making Decisions
- LEGv7 Addressing for 32-bits Literals
- Supporting Functions in Computer Hardware

Representing Instructions

- Instructions are encoded in binary
  - Called machine code

- LEGv7 Pinky instructions
  - Encoded as 16-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!

Hexadecimal

- Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit

0	0000	4	0100	8	1000	C	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

- Example: ECA8
  - 1110 1100 1010 1000

Arithmetic instructions encoding

register instructies	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADDS.N <Rd>,<Rn>,<Rm>	0	0	0	1	1	0	0	Rm			Rn			Rd		
SUBS.N <Rd>,<Rn>,<Rm>	0	0	0	1	1	0	1	Rm			Rn			Rd		

- Instruction fields
  - Opcode ($b_{15} – b_9$): operation code
  - Rm ($b_8 – b_6$): the second register source operand
  - Rn ($b_5 – b_3$): the first register source operand
  - Rd ($b_2 – b_0$): the register destination

Arithmetic instructions encoding

register instructies	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADDS.N <Rd>,<Rn>,<Rm>	0	0	0	1	1	0	0	Rm			Rn			Rd		
SUBS.N <Rd>,<Rn>,<Rm>	0	0	0	1	1	0	1	Rm			Rn			Rd		

- Encode into LEGv7 machine language:

```
ADDS.N   R0, R5, R6
SUBS.N   R1, R7, R6
SUBS.N   R4, R0, R1
```

Load / Store instructions encoding

imm5 instructies	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LDR.N <Rt>, [<Rn>,#<imm7>]	0	1	1	0	1	imm5					Rn			Rt		
STR.N <Rt>, [<Rn>,#<imm7>]	0	1	1	0	0	imm5					Rn			Rt		

- `Rn`: base register
- `Rt`: destination (load) or source (store) register
- `#<imm7>` is a 7-bits unsigned immediate offset (0 t/m 127) that must be dividable by 4
- `imm5 = <imm7> / 4`

Load / Store instructions encoding

imm5 instructies	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LDR.N <Rt>, [<Rn>,#<imm7>]	0	1	1	0	1	imm5					Rn			Rt		
STR.N <Rt>, [<Rn>,#<imm7>]	0	1	1	0	0	imm5					Rn			Rt		

register instructies	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADDS.N <Rd>,<Rn>,<Rm>	0	0	0	1	1	0	0	Rm			Rn			Rd		

- Encode into LEGv7 machine language:

```
LDR.N   R0, [R5, #12]
ADDS.N  R1, R4, R0
STR.N   R1, [R5, #32]
```

Agenda

- Instruction Set
- Operations and Operands
- Representing Instructions in the Computer
- **Logical Operations**
- Instructions for Making Decisions
- LEGv7 Addressing for 32-bits Literals
- Supporting Functions in Computer Hardware

Logical Operations

- Instructions for bitwise manipulation

Operation	C	LEGv7 Pinky
Shift left	<<	`LSLS.N`
Shift right	>> (unsigned)	`LSRS.N`
Bit-by-bit AND	&	`ANDS.N`
Bit-by-bit OR	|	`ORRS.N`
Bit-by-bit EXOR	^	`EORS.N`

- Useful for extracting and inserting groups of bits in a word

Shift Operations

Instructie	Betekenis	N	Z	C	V	Cycles
LSLS.N <Rd>,<Rm>,#<imm5>	<Rd> ← <Rm> << <imm5>	↕	↕	↕	-	1
LSRS.N <Rd>,<Rm>,#<imm5>	<Rd> ← <Rm> >> <imm5>	↕	↕	↕	-	1

- `<imm5>`: (0 – 31) how many positions to shift
- Shift left logical
  - Shift left and fill with `0` bits, shift through C-flag
  - `LSL` by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with `0` bits, shift through C-flag
  - `LSR` by *i* bits divides by 2^i (unsigned only)

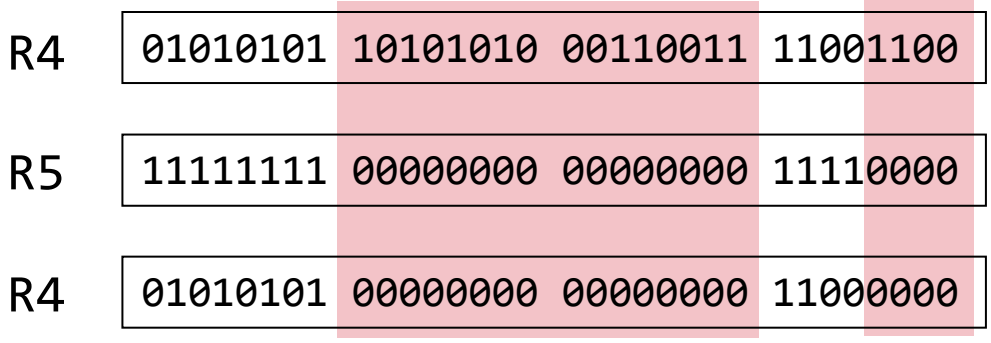
Logical Bitwise Operations

Instructie	Betekenis	N	Z	C	V	Cycles
ANDS.N <Rdn>,<Rm>	<Rdn> ← <Rdn> AND <Rm>	↕	↕	-	-	1
EORS.N <Rdn>,<Rm>	<Rdn> ← <Rdn> EOR <Rm>	↕	↕	-	-	1
ORRS.N <Rdn>,<Rm>	<Rdn> ← <Rdn> OR <Rm>	↕	↕	-	-	1

AND Operations

- Useful to mask bits in a word
  - Set some bits to `0`, leave others unchanged

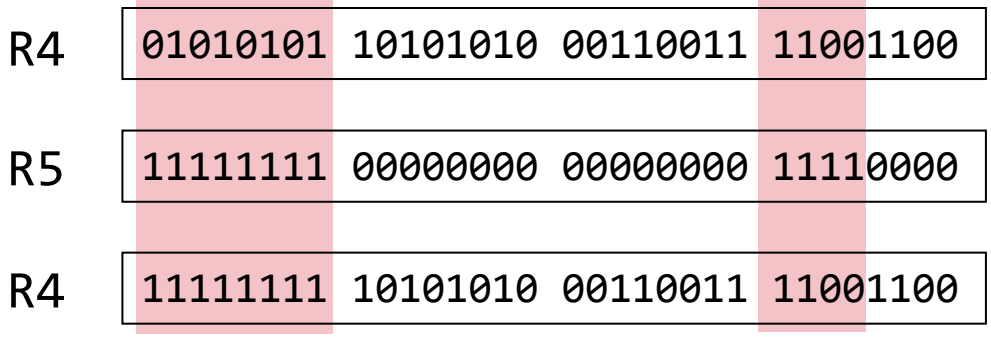
```
ANDS.N R4,R5
```



OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

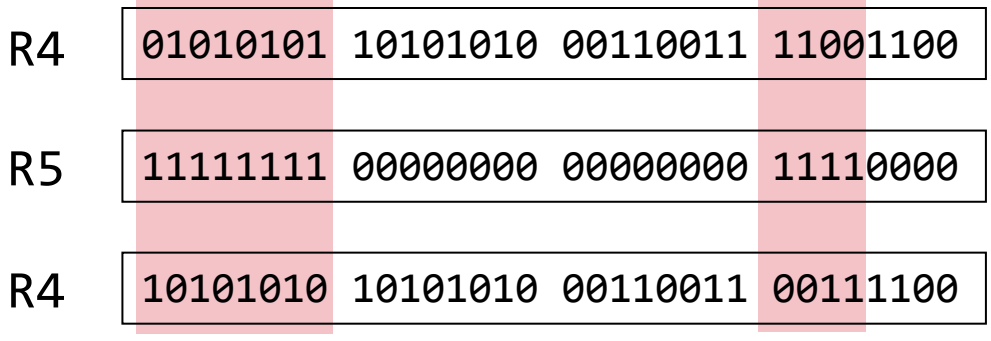
```
ORRS.N R4,R5
```



EOR Operations

- Differencing operation
  - Invert (flip) some bits, leave others unchanged

```
EORS.N R4,R5
```



Shift Example

- C code:

```
// f = (g + h) - (i - h);
f = g + 2*h - i;
```

  - f, g, h, i in R4, R5, R6, R7
- Compiled LEGv7 Pinky code:

```
LSLS.N  R0, R6, #1
ADDS.N  R1, R5, R0
SUBS.N  R4, R1, R7
```

Move Instruction

Instructie	Betekenis	N	Z	C	V	Cycles
MOVS.N <Rd>,#<imm8>	<Rd> ← <imm8>	↕	↕	-	-	1
MOVS.N <Rd>,<Rm>	<Rd> ← <Rm>	↕	↕	-	-	1

`MOVS.N <Rd>, <Rm>`

Is encoded as:

`LSLS.N <Rd>, <Rm>, #0`

imm5 instructies	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LSLS.N <Rd>,<Rm>,#<imm5>	0	0	0	0	0	imm5					Rm			Rd		
MOVS.N <Rd>,<Rm>	0	0	0	0	0	0	0	0	0	0	Rm			Rd		

`MOVS.N <Rd>, <Rm>` is a pseudo instruction
It doesn't have its own machine code

Agenda

- Instruction Set
- Operations and Operands
- Representing Instructions in the Computer
- Logical Operations
- **Instructions for Making Decisions**
- LEGv7 Addressing for 32-bits Literals
- Supporting Functions in Computer Hardware

Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially

CBZ = Compare and Branch if Zero

- `CBZ.N register, L1`
  - if (register == 0) branch forward to instruction labeled `L1:`
- `CBNZ.N register, L1`
  - if (register != 0) branch forward to instruction labeled `L1:`
- `B.N L1`
  - branch unconditionally to instruction labeled `L1:`

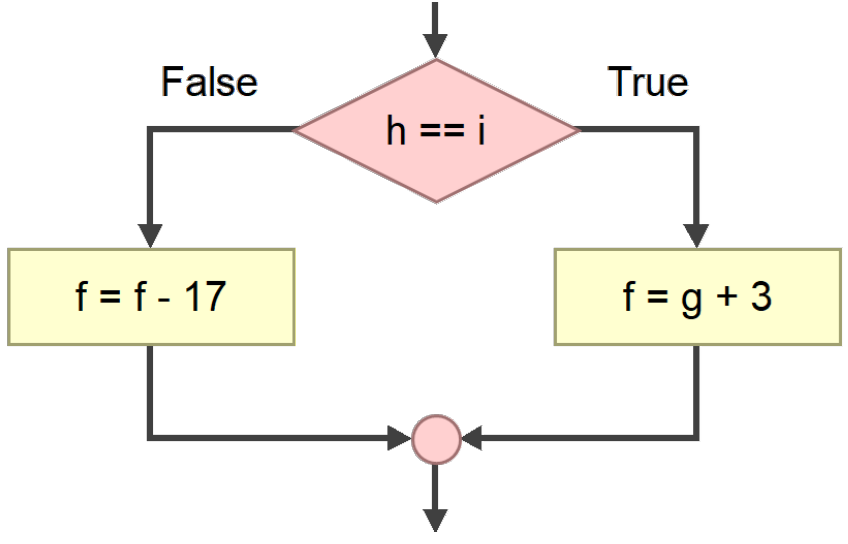
Compiling If Statements

- C code:

```
if (h == i) f = g + 3;
else f = f - 17;
```

  - f, g, h, i in R4, R5, R6, R7
- Compiled LEGv7 Pinky code:

```
        SUBS.N  R0, R6, R7
        CBNZ.N  R0, else
        ADDS.N  R4, R5, #3
        B.N     endif
else:   SUBS.N  R4, #17
endif:
```



Branch is performed by adding an offset to PC

Assembler calculates offsets

Encoding Branch Instructions

Encode the following LEGv7 Pinky code:

```
        SUBS.N  R0, R6, R7
        CBNZ.N  R0, else
        ADDS.N  R4, R5, #3
        B.N     endif
else:   SUBS.N  R4, #17
endif:
```

See Appendix B De LEGv7 architectuur en de Pinky instructieset

Compiling Loop Statements

- C code:

```
int i = 0;
while (save[i] == k) i += 1;
```

  - `i` in R4, k in R5, address of `save` in R6
- Compiled LEGv7 Pinky code:

```
          MOVS.N  R4, #0
while:    LSLS.N  R0, R4, #2
          LDR.N   R1, [R6, R0]
          SUBS.N  R2, R1, R5
          CBNZ.N  R2, endwhile
          ADDS.N  R4, R4, #1
          B.N     while
endwhile:
```

Branch is performed by adding an offset to PC

Assembler calculates offsets

Better Compiling Loop Statements

- C code:

```
int i = 0;
while (save[i] == k) i += 1;
```

  - `i` in R4, k in R5, address of `save` in R6
- Compiled LEGv7 Pinky code:

```
          MOVS.N  R4, #0
while:    LDR.N   R0, [R6, R4]
          SUBS.N  R1, R0, R5
          CBNZ.N  R1, endwhile
          ADDS.N  R4, R4, #4
          B.N     while
endwhile: LSRS.N  R4, #2
```

One less instruction in loop

More Conditional Operations

- Condition codes, set from instruction with S-suffix (ADDS, ANDS, SUBS, ….)
  - negative (N): result had 1 in MSB
  - zero (Z): result was 0
  - overflow (V): result sign overflowed
  - carry (C): result had carryout from MSB
- Use `CMP.N`, then conditionally branch:
  - **B.EQ** (equal)
  - **B.NE** (not equal)
  - **B.LT** (less than, < signed), **B.LO** (lower, < unsigned)
  - **B.LE** (less than or equal, ≤ signed), **B.LS** (lower or same, ≤ unsigned)
  - **B.GT** (greater than, > signed), **B.HI** (higher, > unsigned)
  - **B.GE** (greater than or equal, ≥ signed), **B.HS** (higher or same, ≥ unsigned)

`CMPS.N` would have been a better name

Signed versus Unsigned

- Signed comparison ≠ Unsigned comparison
- Example
  - R0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - R1 = 0000 0000 0000 0000 0000 0000 0000 0001
  - R0 < R1 signed
    - −1 < +1
  - R0 > R1 unsigned
    - +4294967295 > +1

B<c>.N condities

cond	<c>	betekenis	conditie	betekenis na CMP Rn,op2
0000	EQ	Equal	Z == 1	Rn == op2
0001	NE	Not Equal	Z == 0	Rn != op2
0010	CS / LO	Carry Set / LOwer	C == 1	Rn < op2, unsigned
0011	CC / HS	Carry Clear / Higher or Same	C == 0	Rn >= op2, unsigned
0100	MI	MInus	N == 1	
0101	PL	Plus	N == 0	
0110	VS	Overflow Set	V == 1	
0111	VC	Overflow Clear	V == 0	
1000	HI	Higher	C == 1 and Z == 0	Rn > op2, unsigned
1001	LS	Lower or Same	C == 0 or Z == 1	Rn <= op2, unsigned
1010	GE	Greater than or Equal	N == V	Rn >= op2, signed
1011	LT	Less Than	N != V	Rn < op2, signed
1100	GT	Greater Than	Z == 0 and N == V	Rn > op2, signed
1101	LE	Less than or Equal	Z == 1 or N != V	Rn <= op2, signed
1110	None / AL	ALways (unconditional)	True	

Conditional Example

- C code:

```
int a, b;
if (a > b) a += 1;
```

  - a, b in R4, R5
- Compiled LEGv7 Pinky code:

```
    CMP.N   R4, R5
    BLE.N   exit
    ADDS.N  R4, #1
exit:
```

Note: condition in B<c>.N is complement of condition in `if`

Conditional Example

- C code:

```
unsigned int a, b;
if (a > b) a += 1;
```

  - a, b in R4, R5
- Compiled LEGv7 Pinky code:

```
    CMP.N   R4, R5
    BLS.N   exit
    ADDS.N  R4, #1
exit:
```

Agenda

- Instruction Set
- Operations and Operands
- Representing Instructions in the Computer
- Logical Operations
- Instructions for Making Decisions
- **LEGv7 Addressing for 32-bits Literals**
- Supporting Functions in Computer Hardware

Load 32-bit literal in register

- `MOVS.N Rd, #<imm8>` can be used to load an 8-bits unsigned constant in Rd.
- How to load a 32-bits constant in Rd with a 16-bits instruction?
- Literal addressing mode:
  - `LDR.N  R6, =0x1234567`
- Is encoded as:
  -
```
LDR.N  R6, [PC, #offset]
//...
.word  0x1234567 // literal section
```

Instead of a constant, a `label` can be used (to load an address)

Note: offset is 10-bits unsigned which should be dividable by 4

Agenda

- Instruction Set
- Operations and Operands
- Representing Instructions in the Computer
- Logical Operations
- Instructions for Making Decisions
- LEGv7 Addressing for 32-bits Literals
- **Supporting Functions in Computer Hardware**

Subroutine (Function) Calling

Steps required:

- Place parameters in registers `R0` to `R3`
- Transfer control to function
- Acquire local storage for function
- Perform function's operations
- Place result in register `R0` for caller
- Return to place of call (address in LR)

Function Call Instructions

- Function call: Branch with Link and eXchange

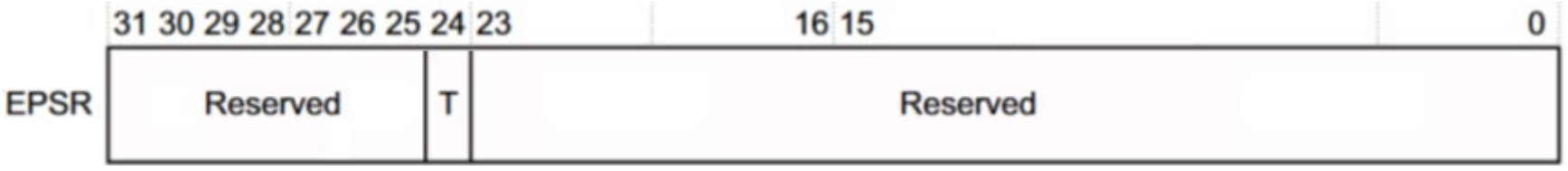
```
LDR.N   R4, =FunctionLabel
BLX.N   R4
```

  - Address of following instruction put in LR (Link Register)
  - Jumps to target address
- Function return: Branch and eXchange

```
BX.N    LR
```

- Copies LR to program counter
  - Can also be used for computed jumps
    - e.g., for case/switch statements or jump to function pointers

T-bit



- When the PC is loaded (e.g. by BLX or BX) bit 0 is copied to the T-bit.
- This way the processor can switch (eXchange) between the Thumb and ARM instruction sets.
- ARM Cortex-M only supports Thumb and raises an exception when T is 0.

Register Usage

- R0 to R3: temporary registers
  - Used to pass arguments to functions and pass return value
  - Not preserved by the callee

Callee = function which is called

- R4 to R7: saved registers
  - If used, the callee saves and restores them

ARM Call Convention

Should we follow this convention?

Leaf Function Example

- C code:

```
int leaf_example(int g, int h, int i, int j) {
    int f;
    f = (g + h) - (i - j);
    return f;
}
```

- Compiled LEGv7 Pinky code:

```
        .thumb_func
leaf_example:
        ADDS.N  R0, R0, R1
        SUBS.N  R2, R2, R3
        SUBS.N  R0, R0, R2
        BX.N    LR
```

Leaf Function Example

- C code:

```
int leaf_example(int g, int h, int i, int j) {
    int f;
    f = (g + h) - (i - j);
    return f;
}
```

- Compiled LEGv7 Pinky code:

```
        .thumb_func
leaf_example:
        ADDS.N  A1, A1, A2
        SUBS.N  A3, A3, A4
        SUBS.N  A1, A1, A3
        BX.N    LR
```

R0 to R3 may also be called A1 to A4

Leaf Function Call Example

- C code:

```
int a, b, c, d;
//...
a = leaf_example(b, c, d, c);
```

  - a to d in R4 to R7

- Compiled LEGv7 Pinky code:

```
MOVS.N  R0, R5
MOVS.N  R1, R6
MOVS.N  R2, R7
MOVS.N  R3, R6
LDR.N   R4, =leaf_example
BLX.N   R4
MOVS.N  R4, R0
```

Leaf Function Call Example

- C code:

```
int a, b, c, d;
//...
a = leaf_example(b, c, d, c);
```

  - a to d in R4 to R7 (V1 to V4)

- Compiled LEGv7 Pinky code:

```
MOVS.N  A1, V2
MOVS.N  A2, V3
MOVS.N  A3, V4
MOVS.N  A4, V3
LDR.N   V1, =leaf_example
BLX.N   V1
MOVS.N  V1, A1
```

R4 to R7 may also be called V1 to V4

Leaf Function Example

- C code:

```
int leaf_example(int g, int h, int i, int j) {
    int f = (g + h) - (i + j);
    // more code which needs g, h, i and j
    return f;
}
```

- Compiled LEGv7 Pinky code:

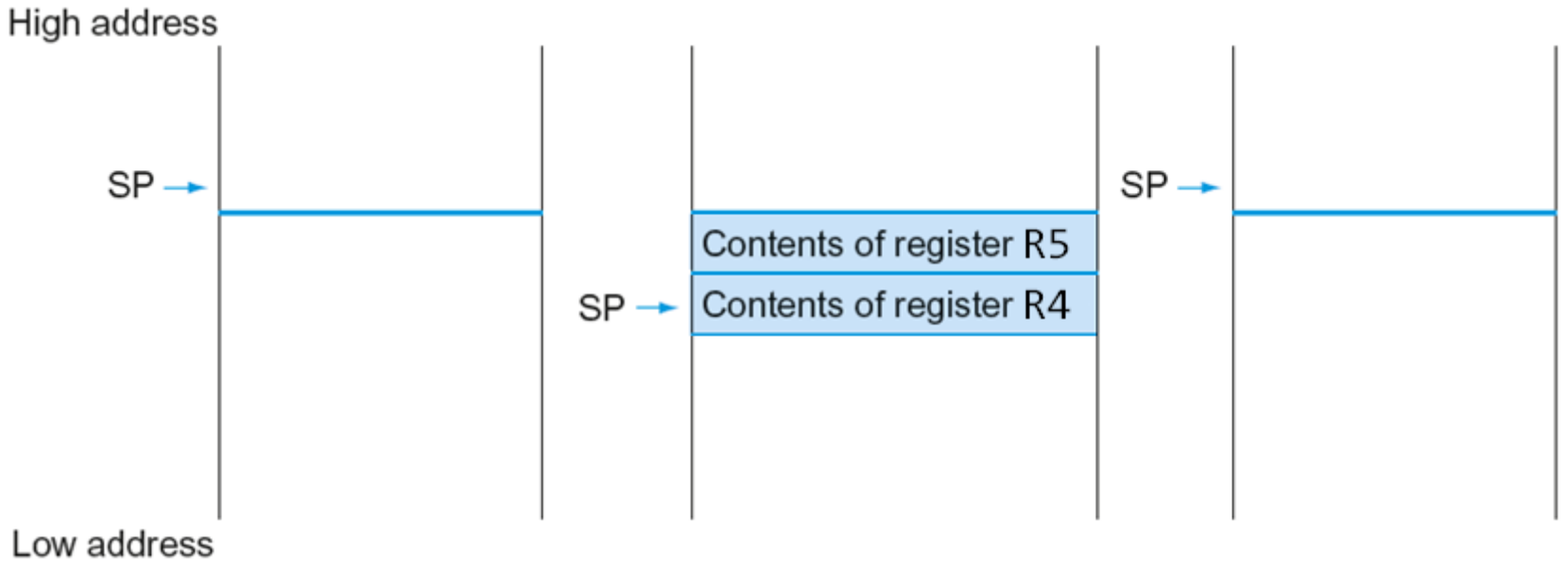
What's wrong?

Does not conform to ARM ABI

Where to temporally store R4 and R5?

```
        .thumb_func
leaf_example:
        ADDS.N  R4, R0, R1
        SUBS.N  R5, R2, R3
// more code which needs R0, R1, R2 and R3
        SUBS.N  R0, R4, R5
        BX.N    LR
```

Local Data on the Stack



Before function call | During function call | After function call

Leaf Function Example

- Compiled LEGv7 Pinky code:

```
        .thumb_func
leaf_example:
        PUSH.N  {R4, R5}
        ADDS.N  R4, R0, R1
        SUBS.N  R5, R2, R3
// more code which needs R0, R1, R2 and R3
        SUBS.N  R0, R4, R5
        POP.N   {R4, R5}
        BX.N    LR
```

Leaf Function Example

- C code:

```
unsigned int gcd(unsigned int a, unsigned int b) {
    while (a != b) {
        if (a > b) {
            a = a – b;
        }
        else {
            b = b – a;
        }
    }
    return a;
}
```

https://en.wikipedia.org/wiki/Euclidean_algorithm

- Compiled LEGv7 Pinky code:

Leaf Function Example

- Compiled LEGv7 Pinky code:

```
        .thumb_func
gcd:    CMP.N   R0, R1
        BEQ.N   end
        BLS.N   else
        SUBS.N  R0, R0, R1
        B.N     gcd
else:   SUBS.N  R1, R1, R0
        B.N     gcd
end:    BX.N    LR
```

Non-Leaf Function Example

- C code:

```
bool is_coprime(unsigned int a, unsigned int b) {
    return gcd(a, b) == 1;
}
```

https://en.wikipedia.org/wiki/Coprime_integers

- Compiled LEGv7 Pinky code:

```
        .thumb_func
is_coprime:
        LDR.N   R2, =gcd
        PUSH.N  {LR}
        BLX.N   R2
        CMP.N   R0, #1
        BEQ.N   end
        MOVS.N  R0, #0
end:    POP.N   {PC}
```

Encoding Functions

Encode the following LEGv7 Pinky code:

```
        LDR.N   R4, =leaf_example
        BLX.N   R4
        MOVS.N  R4, R0
end:    B.N     end
        .thumb_func
leaf_example:
        ADDS.N  R0, R0, R1
        SUBS.N  R2, R2, R3
        SUBS.N  R0, R0, R2
        BX.N    LR
```

See Appendix B De LEGv7 architectuur en de Pinky instructieset

Decode

Decode the following LEGv7 Pinky machine code

```
0110100000000011
0110100000001010
0110000000000010
0110000000001011
0100011101110000
```

Can you guess which function this is?

See Appendix C De LEGv7 architectuur en de Pinky instructieset

Decode

Decode the following LEGv7 Pinky machine code

```
1bf0
b908
1cec
e001
002c
3c11
```

See Appendix C De LEGv7 architectuur en de Pinky instructieset

Recursive Function Example

- C code:

```
unsigned int fib(unsigned int n) {
    if (n < 2) {
        return n;
    }
    return fib(n-1) + fib(n-2);
}
```

- Compiled LEGv7 Pinky code:

https://en.wikipedia.org/wiki/Fibonacci_number

Recursive Function Example


```
        .thumb_func
fib:    CMP.N  R0, #2
        BHS.N  endif
        BX.N   LR
endif:  PUSH.N {R4, R5, R6, LR}
        LDR.N  R4, =fib
        MOVS.N R5, R0
        SUBS.N R0, #1
        BLX.N  R4
        MOVS.N R6, R0
        MOVS.N R0, R5
        SUBS.N R0, #2
        BLX.N  R4
        ADDS.N R0, R0, R6
        POP.N  {R4, R5, R6, PC}
```

372135 clock cycles to calculate `fib(20)`

Recursive Function Example


```
        .thumb_func
fib:    LDR.N  R1, =fib_r
        .thumb_func
fib2_r: CMP.N  R0, #2
        BHS.N  endif
        BX.N   LR
endif:  PUSH.N {R0, R4, LR}
        SUBS.N R0, #1
        BLX.N  R1
        MOVS.N R2, R0
        LDR.N  R0, [SP, #0]
        SUBS.N R0, #2
        BLX.N  R1
        ADDS.N R0, R0, R2
        POP.N  {R2, R4, PC} // R2 is dummy
```

328357 clock cycles to calculate `fib(20)`

Better Non-Recursive fib(n)


```
        .thumb_func
fib:    CMP.N  R0, #2
        BLO.N  end
endif:  SUBS.N R3, R0, #1
        MOVS.N R2, #0
        MOVS.N R1, #1
loop:   ADDS.N R0, R1, R2
        MOVS.N R2, R1
        MOVS.N R1, R0
        SUBS.N R3, #1
        BNE.N  loop
end:    BX.N   LR
```

121 clock cycles to calculate `fib(20)`

Geef de C code voor unsigned int `fib(unsigned int n)`

Volgende les…

Theorie:

- Hardware implementatie van een vereenvoudigde versie van de LEGv7-M architectuur
- Deze implementatie versnellen door het toepassen van:
  - een pipeline
  - branch prediction
  - cache geheugen

Lab:

- Aansturen van leds op verschillende abstractieniveaus.

Aan de slag!

Aan de slag met [Opdrachten_Week_1.pdf]