ARM architecture (armarch)

Teil 4

Armarch - mach(ine) lang(uage) I/II

- Assembly language kann von Menschen gelesen werden
- Die digitale Hardware wie der Prozessor kennt aber nur 1 und 0
- Daher muss ein Assembler Programm in machine language übersetzt werden
- ARM verwendet 32bit instructions
- Auch wenn nicht alle instructions 32bit brauchen, sind alle instructions gleich lang
  - ->regularity supports simplicity
  - wenn nicht alle instructions gleich lang wären würde das zu größerer Komplexität führen
- Die simplicity würde ein einzelnes instruction format fordern, aber das wäre zu restriktiv
  - -> Design Principle 4: Good design demands good compromises

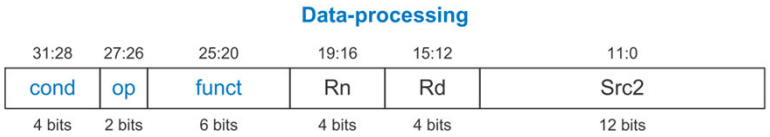
Armarch - mach lang I/II

- ARM macht den Kompromiss drei instruction formats zu definieren: data-processing, memory und branch
- Die kleine Anzahl gibt etwas regularity und damit einfachere (decoder) Hardware, wobei aber trotzdem die Anforderungen der verschiedenen instructions nachgekommen wird
- data processing instructions haben ein src1 register operand, ein src2 operand, der entweder immediate oder register ist, welches optional geshiftet werden kann und einen destination operand
  - data processing instructions haben mehrere Varianten für den src2 operand
- memory instructions haben drei operanden: base register operand, offset operand (immediate oder optional shifted register) und ein weiteres register (destination operand für LDR oder source für STR)
- branch instructions haben einen 24-bit immediate offset

Armarch - mach lang - data processing instr

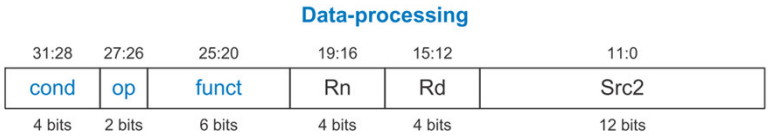
- Das data processing instruction format ist das am meisten genutzte format
- Beispiel: SUBS R1,R9, #11 @<com> <rdest>, <src1>, <src2>
  - Der src1 operand ist ein register
  - Der src2 operand ist entweder immediate oder register, welches optional geshiftet werden kann
  - Der destination operand ist ein register

Armarch - mach lang - data processing instr - instruction format I/II



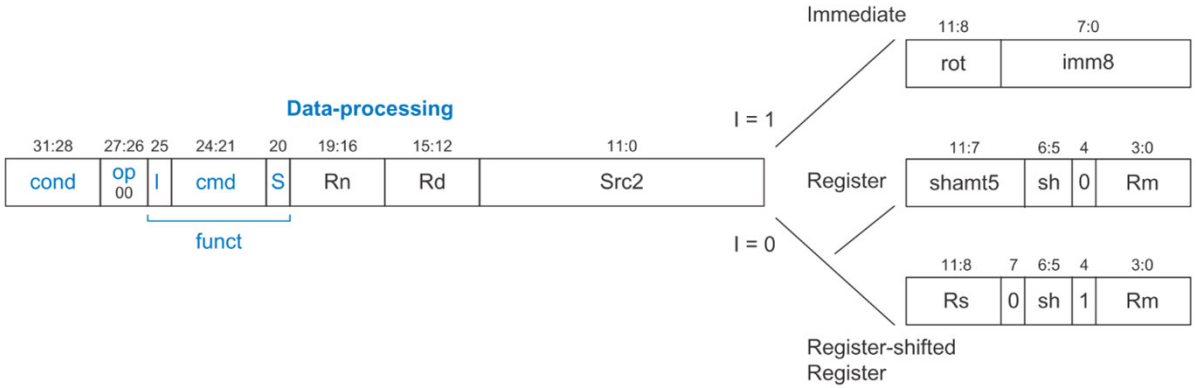
- Die 32bit data processing instructions werden in sechs Teile aufgeteilt: cond, op, funct, Rn, Rd und Src2
- Die eigentliche operation wird durch die blauen Teile festgelegt:
  - op= op (auch opcode oder operation code) genannt; 0b00 für data processing instructions
  - func= functional code
  - cond= conditional execution (basierend auf den conditional flags!!!)
    - cond= 0b1110 steht für eine unconditional instruction

Armarch - mach lang - data processing instr - instruction format II/II



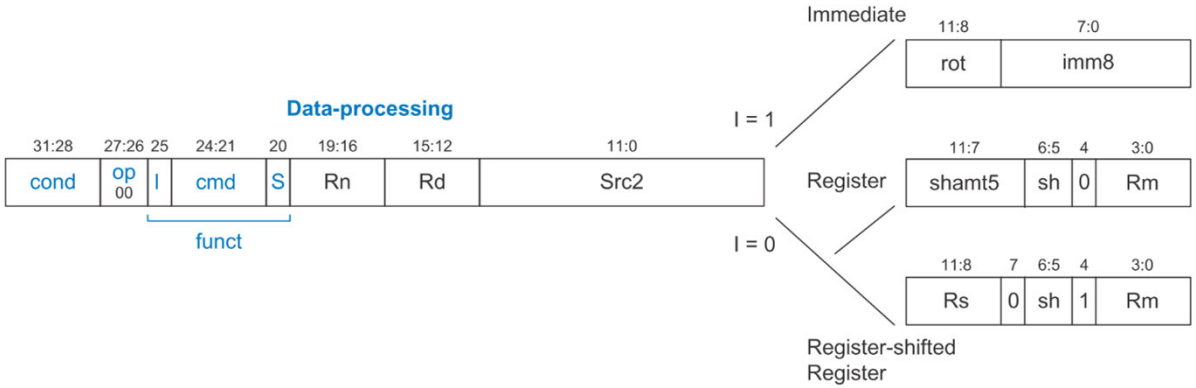
- Die operands werden durch Rn, Rd und Src2 festgelegt
- Rn ist das first source register
- Src2 ist der second source operand (register oder imm)
- Rd ist das destination register

Armarch - mach lang - data processing instr - funct details



- Das funct bitfield hat 3 Teile: “I”, “cmd” und “S”
  - I ist 1 wenn Src2 ein immediate ist
  - S ist 1, wenn die instruction die condition flags setzt (S=set)
    - Beispiel: SUBS R1,R9, #11
  - cmd kennzeichnet die spezifische data processing instruction
    - Beispiel: ADD -> cmd= 0b0100 bzw. SUB -> cmd=0b0010

Armarch - mach lang - data processing instr - Src2 variations



- Es gibt drei Varianten des Src2 encodings
  - (1) immediate, (2) register (Rm) optional geshifted durch eine constant (shamt5) und (3) register (Rm) gehifted durch ein anderes register (Rs)
  - sh kodiert den Typ der shift-operation (vgl. später)
  - -> schauen wir uns zuerst (2) an, da das am einfachsten ist

Armarch - mach lang - data processing instr - Src2 variations - (2) register (without shift) - example

- Beispiel: Befehle mit 3 register operands (ohne shift)
  - Die Umwandlung von ASM in machine code ist am einfachsten, wenn man die Werte von jedem bitfield ausschreibt, diese in binär umwandelt und dann schließlich alles in HEX umwandelt

Assembly Code

```
ADD R5, R6, R7
    (0xE0865007)
SUB R8, R9, R10
    (0xE049800A)
```

Field Values

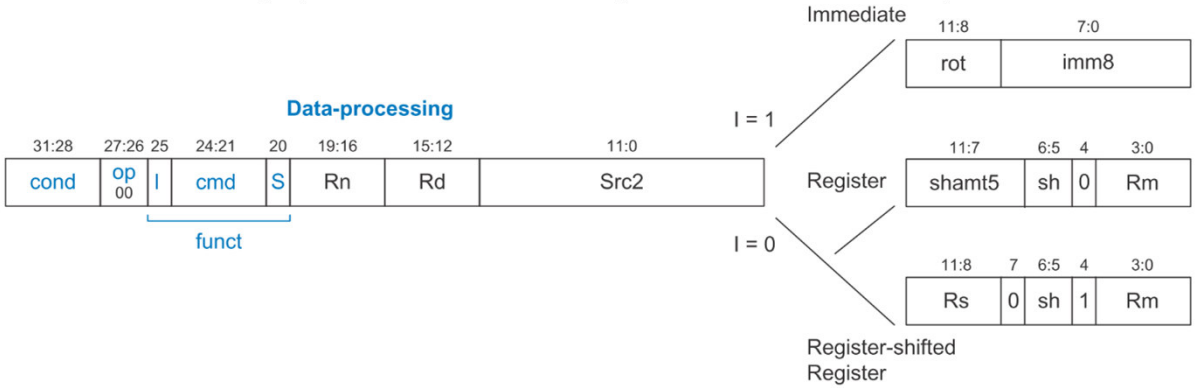
31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
1110_2	00_2	0	0100_2	0	6	5	0	0	0	7
1110_2	00_2	0	0010_2	0	9	8	0	0	0	10
cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm

Machine Code

31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
1110	00	0	0100	0	0110	0101	00000	00	0	0111
1110	00	0	0010	0	1001	1000	00000	00	0	1010
cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm

  - Anm: Achtung!
    - Das destination register ist das erste register in ASM aber das zweite im encoding!
    - Rn und Rm sind der erste bzw. zweite source operand in ASM!

Armarch - mach lang - data processing instr - Src2 variations - (1) immediate (without shift)



- zu (1) immediate
  - Die Repräsentation des immediate Werts ist ungewöhnlich
  - Der 8bit unsigned immediate Wert (imm8) wird genommen und mit dem zweifachen 4-bit rotation Wert (rot) nach rechts rotiert um eine 32bit constant zu erzeugen:
    resultconst=imm8 right rotate by 2xrot

Armarch - mach lang - data processing instr - Src2 variations - (1) immediate - example rot calc (B)

- resultconst=imm8 right rotate by 2xrot
  - Beispiel: Immediate rotations und resulting 32-bit constant für imm8 = 0xFF

rot	32-bit Constant
0000	0000 0000 0000 0000 0000 0000 1111 1111
0001	1100 0000 0000 0000 0000 0000 0011 1111
0010	1111 0000 0000 0000 0000 0000 0000 1111
…	…
1111	0000 0000 0000 0000 0000 0011 1111 1100

Armarch - mach lang - data processing instr - Src2 variations - (1) immediate (without shift) - example

- Beispiel: Immediate und 2 register operands

Assembly Code

```
ADD R0, R1, #42
    (0xE281002A)
SUB R2, R3, #0xFF0
    (0xE2432EFF)
```

Field Values

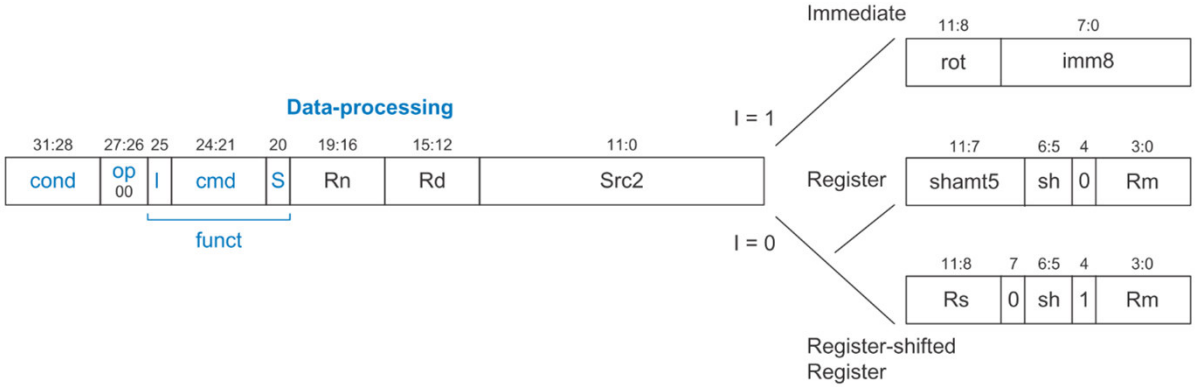
31:28	27:26	25	24:21	20	19:16	15:12	11:8	7:0
1110_2	00_2	1	0100_2	0	1	0	0	42
1110_2	00_2	1	0010_2	0	3	2	14	255
cond	op	I	cmd	S	Rn	Rd	rot	imm8

Machine Code

31:28	27:26	25	24:21	20	19:16	15:12	11:8	7:0
1110	00	1	0100	0	0001	0000	0000	00101010
1110	00	1	0010	0	0011	0010	1110	11111111
cond	op	I	cmd	S	Rn	Rd	rot	imm8

- Anm: Achtung!
  - Das destination register ist das erste register in ASM aber das zweite im encoding!
  - Rn ist der erste source operand in ASM!
  - Der immediate 42 kann direkt in imm8 kodiert werden
  - Der immediate 0xFF0 jedoch nicht -> rol 4bit ->ror um 28bits -> rot=14

Armarch - mach lang - data processing instr - Src2 variations - (3) register-shifted register



- zu (3) register-shifted register (register (Rm) gehifted durch ein anderes register (Rs))
  - Diese Variante spielt hauptächlich eine Rolle bei shifts
  - -> shifts

Armarch - mach lang - data processing instr - shifts

- Shift-operations sind auch data-processing instructions
- Es gibt zwei Varianten: shift mit shift amount als immeditate (format register (2)) und shift mit shift amount als register (format register-shifted-register (3))
- Alle shift instructions haben 0b1101 als cmd-bitfield (oder auch 0b1111 vgl. Befehlsliste)
- Das sh-bitfield legt den Type von shift-operation fest

Instruction	sh	Operation
LSL	00_2	Logical shift left
LSR	01_2	Logical shift right
ASR	10_2	Arithmetic shift right
ROR	11_2	Rotate right

- Rm beinhaltet den Wert, der geshifted werden soll
- Das Ergebnis wird in Rd platziert

Armarch - mach lang - data processing instr - shifts - example (format 2) (immediate shift amounts)

Assembly Code

```
LSL  R0, R9, #7
     (0xE1A00389)
ROR  R3, R5, #21
     (0xE1A03AE5)
```

Field Values

31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
1110_2	00_2	0	1101_2	0	0	0	7	00_2	0	9
1110_2	00_2	0	1101_2	0	0	3	21	11_2	0	5
cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm

Machine Code

31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
1110	00	0	1101	0	0000	0000	00111	00	0	1001
1110	00	0	1101	0	0000	0011	10101	11	0	0101
cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm

Armarch - mach lang - data processing instr - shifts - example (format 3) (register shift amounts)

Assembly Code

```
LSR R4, R8, R6
(0xE1A04638)
ASR R5, R1, R12
(0xE1A05C51)
```

Field Values

31:28	27:26	25	24:21	20	19:16	15:12	11:8	7	6:5	4	3:0
1110_2	00_2	0	1101_2	0	0	4	6	0	01_2	1	8
1110_2	00_2	0	1101_2	0	0	5	12	0	10_2	1	1
cond	op	I	cmd	S	Rn	Rd	Rs		sh		Rm

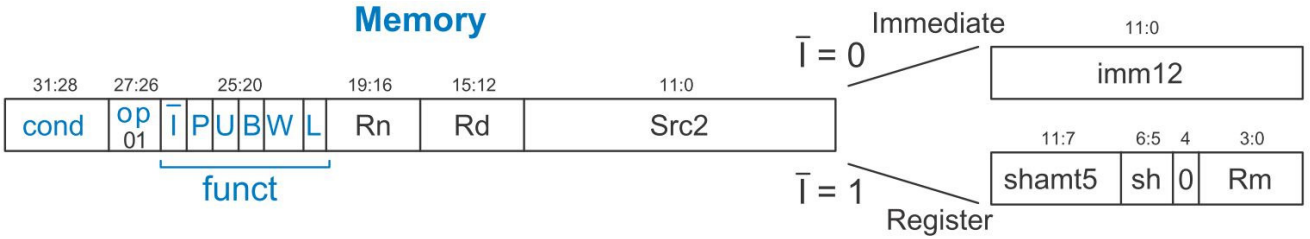
Machine Code

31:28	27:26	25	24:21	20	19:16	15:12	11:8	7	6:5	4	3:0
1110	00	0	1101	0	0000	0100	0110	0	01	1	1000
1110	00	0	1101	0	0000	0101	1100	0	10	1	0001
cond	op	I	cmd	S	Rn	Rd	Rs		sh		Rm

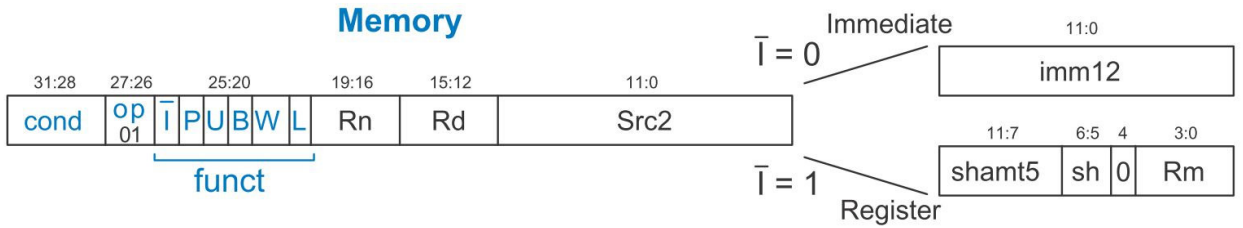
- Rs bzw. die unteren 8 bits von Rs legt den shift amount fest
  - Beispiel: Rs= 0xF001001C -> nur 0x1C
- Anm:
  - Ein shift von 31 schiebt alle Bits raus!
  - Ein rotate ist zyklisch -> rotate von 50 -> 50 mod 32= 18

Armarch - mach lang - memory instr I/II

- memory instructions benutzen ein ähnliches aber nicht dasselbe format wie data processing instructions
- Die 32bit memory instructions werden auch in sechs Teile aufgeteilt: cond, op, funct, Rn, Rd und Src2



Armarch - mach lang - memory instr II/II



- memory instructions benutzen ein anderes funct-bitfield
- op hat den Wert 0b01
- Rn ist das base register
- Src2 bestimmt den offset
- Rd ist bei einem load das destination register und bei einem store das source register
- Es gibt zwei Varianten für das Src2-bitfiled: (1) 12bit immediate oder (2) register (Rm) das optional geshifted werden kann

Armarch - mach lang - memory instr - funct details

- funct besteht aus sechs control bits: !I, P, U, B, W und L
  - !I (immeditae) und U (add) bestimmen ob der offset ein immediate oder register ist und ob er subtrahiert oder addiert werden soll
  - P (pre-index) und W (writeback) spezifizieren den index mode

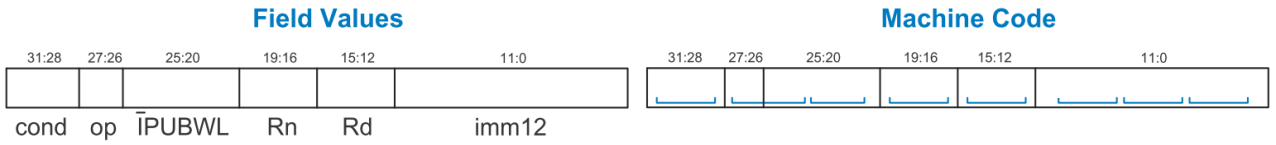
P	W	Index Mode
0	0	Post-index
0	1	Not supported
1	0	Offset
1	1	Pre-index

- L (load) und B (byte) legen den Typ der memory operation fest (load/store und 32bit bzw. 8bit size)

L	B	Instruction
0	0	STR
0	1	STRB
1	0	LDR
1	1	LDRB

Armarch - mach lang - memory instr - Übung

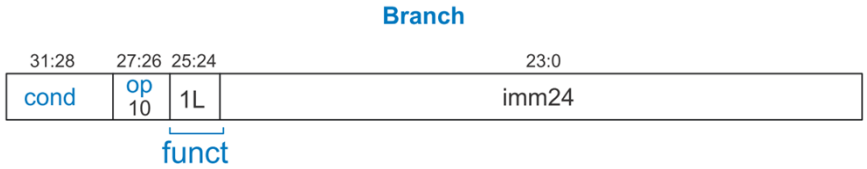
- Übersetzen Sie folgenden ASM-Code in machine code
  - `STR R11, [R5], #-26 @post indexing!`
- Lösung:



Teil 5

Armarch - mach lang - branch instr

- Branch instructions benutzen einen 24-bit immediate operand (imm24)



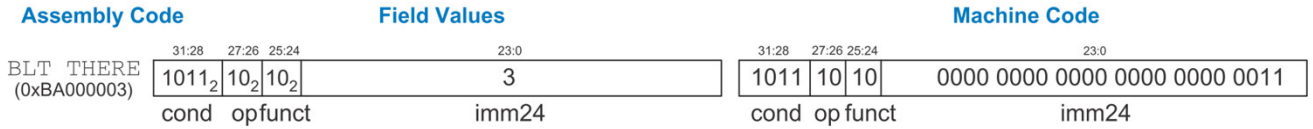
- Das op bitfield ist 10
- Das funct bitfield hat nur 2bits
  - Das obere bit ist immer 1
  - Das untere bit L bestimmt den typ des branches (L=1 -> BL; L=0 ->B)
- Der imm24 wird benutzt die branch target address (BTA) zu bestimmen:
  BTA=(PC+8)+(imm24<<2) -> imm24*4= BTA - (PC + 8)
  - Bedeutung: der um zwei Stellen nach links geshiftete imm24 Wert legt die „Differenz“ der BTA zu (PC+8) fest

Armarch - mach lang - branch instr - example

- ASM-Code

```
0x80A0 BLT THERE
0x80A4 ADD R0, R1, R2
0x80A8 SUB R0, R0, R9
0x80AC ADD SP, SP, #8
0x80B0 MOV PC, LR
0x80B4 THERE:
         SUB R0, R0, #1
0x80B8 ADD R3, R3, #0x5
```

- Die BLT instruction hat eine BTA von 0x80B4 (Adresse des labels THERE)
- imm24*4= BTA - (PC + 8); -> imm24*4=0x80B4-(0x80A0+8) -> imm24*4=0xC -> imm24=0xC/4=0x3



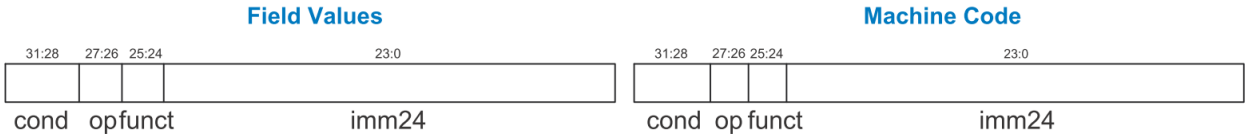
Armarch - mach lang - branch instr - Übung

- Berechnen Sie das immediate-bitfield und geben Sie den machine code für die branch instruction in folgendem ASM-Programm an
- Falls Sie negativen Wert für imm24 bekommen, kodieren Sie Ihn im Zweierkomplement

- ASM-Code

```
0x8040 TEST:
       LDRB R5, [R0, R3]
0x8044 STRB R5, [R1, R3]
0x8048 ADD R3, R3, #1
0x8044 MOV PC, LR
0x8050 BL TEST
0x8054 LDR R3, [R1], #4
0x8058 SUB R4, R3, #9
```

- Rechnung



-> hex:

Armarch - mach lang - summary addressing modes I/II

- ARM verfügt im Wesentlichen über vier addressing modes: register, immediate, base und PC-relative
- register, immediate und base wird für lesende und schreibende operation verwendet
- PC-relative zum Schreiben des Programm Counters (PC)

Operand Addressing Mode	Example	Description
Register		
Register-only	`ADD R3, R2, R1`	R3 ← R2 + R1
Immediate-shifted register	`SUB R4, R5, R9, LSR #2`	R4 ← R5 − (R9 >> 2)
Register-shifted register	`ORR R0, R10, R2, ROR R7`	R0 ← R10 | (R2 ROR R7)
Immediate	`SUB R3, R2, #25`	R3 ← R2 − 25
Base		
Immediate offset	`STR R6, [R11, #77]`	mem[R11+77] ← R6
Register offset	`LDR R12, [R1, −R5]`	R12 ← mem[R1 − R5]
Immediate-shifted register offset	`LDR R8, [R9, R2, LSL #2]`	R8 ← mem[R9 + (R2 << 2)]
PC-Relative	`B LABEL1`	Branch to `LABEL1`

Armarch - mach lang - summary addressing modes II/II

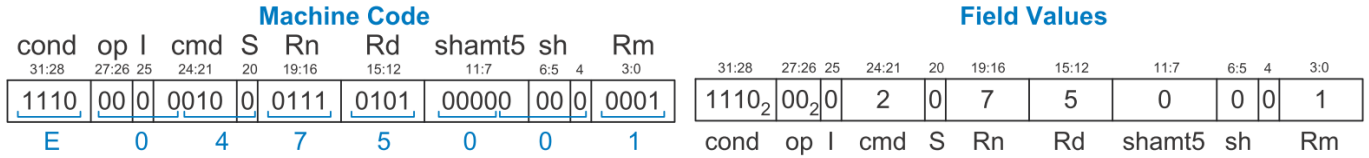
- Data-processing instructions verwenden register oder immediate addressing
  - first source operand ist register; second source operand ist register oder immediate
  - Wenn der second source operand ein register ist kann dieser optional durch einen immediate oder ein weiteres register geschiftet werden
- Memory instructions verwenden base addressing
  - Die base address kommt aus einem register und der offset kann ein immediate, ein register oder ein geshiftetes registers ein
- Branch instructions verwenden PC-relative addressing
  - BTA= offset+ PC+ 8

Armarch - mach lang - interpret mach lang

- Um machine code zu interpretieren muss man die bitfieds der 32bit instructions entziffern
- Die instruction formats sind unterschiedlich, aber alle starten mit dem 4bit cond field und dem 2bit op field
- Am besten betrachtet man zunächst das op field:
  - wenn 0b00-> data processing; 0b01-> memory instruction; 0b10 -> branch instruction

Armarch - mach lang - interpret mach lang - example

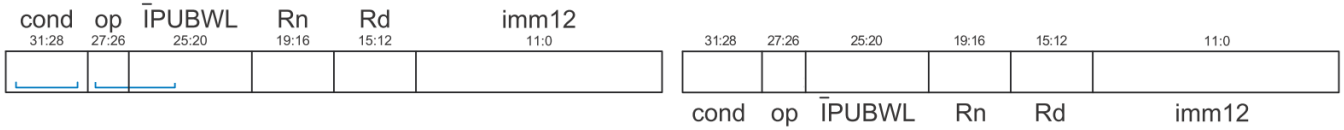
- Übersetzen Sie den folgenden machine code in ARM assembly code
  - 0xE0475001
- Lösung:



  - op(bit27:26)=0b00 -> data processing;
  - cmd(bit24:21)=0b0010; I(bit25)=0; -> SUB mit Src2=register
  - Rd=5, Rn=7, Rm=1
  - -> instr: SUB R5, R7, R1

Armarch - mach lang - interpret mach lang - Übung

- Übersetzen Sie den folgenden machine code in ARM assembly code
  - 0xE5949010
- Lösung:

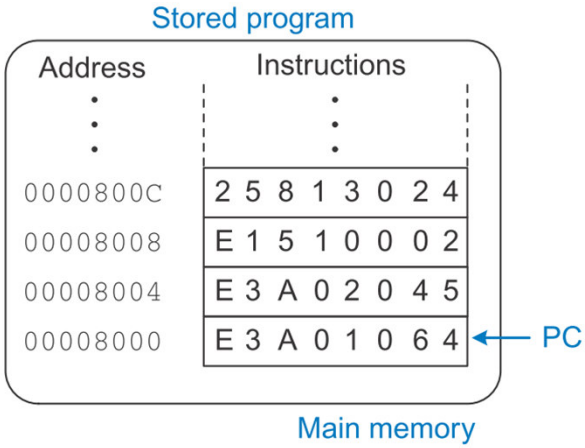


- -> instr:

Armarch - mach lang - stored program (B)

- Ein Programm in machine language ist eine Sequenz von 32bit Zahlen, die instructions repräsentieren

Assembly code	Machine code
MOV R1, #100	0xE3A01064
MOV R2, #69	0xE3A02045
CMP R1, R2	0xE1510002
STRHS R3, [R1, #0x24]	0x25813024



Armarch - memory - details behind „=<label>“

- Viele Programme brauchen 32bit constants (literals) oder Adressen
- `MOV` akzeptiert nur 12bit immediate
- `LDR` kann nur address offsets von 12bit verarbeiten
- -> Vorgehen:
  - Die Variablen / Konstanten werden wie gewohnt in den entsprechenden segment angelegt (.rodata, .data, .bss)
  - Im .text segment in der Nähe des gerade auszuführenden Codes wird die komplette address des gewünschten datums abgelegt (address pool)
  - Ein erster `LDR` läd nun diese address in ein register
  - Ein zweiter `LDR` nutzt nun dieses register als pointer und läd dann den eigentlichen Wert (in ein register)
    -> nun hat man den vollständigen Wert

OSTBAYERISCHE TECHNISCHE HOCHSCHULE REGENSBURG

Armarch - memory - details behind „=<label>“ - example

- C-Code

```
int length=6;
int height=5;
int res;

int mycfn()
{
 res=height+length;
 return res;
}

int main(){
...
}
```

- ASM-Code

```
.section .data
  .align  2
LANCHOR0:
height:
  .int  5
length:
  .int  6
.section .bss
  .align 2
  .comm res,4
.section .text
mycfn:
  ldr r2, .adrpl   @ load addres of anchor to .data from
                   @ address pool
  ldr r3, [r2]     @ load height
  ldr r0, [r2, #4] @ load length
  add r0, r3, r0
  ldr r3, .adrpl+4 @ load address of res from address pool
  str r0, [r3]     @ store height+length in res
  bx  lr           @ return
  .align  2
.adrpl:
  .int  .LANCHOR0
  .int  res
```

Armarch - memory - details behind „=<label>“ - example simplified

- Dieses Vorgehen mit dem address pooling kann vereinfacht werden, in dem man `ldr <rd>, =<label>` verwendet

- ASM-Code

```
.section .data
  .align  2
LANCHOR0:
height:
  .int  5
length:
  .int  6
.section .bss
  .align 2
  .comm res,4
.section .text
mycfn:
  ldr r2, .adrpl
  ldr r3, [r2]
  ldr r0, [r2, #4]
  add r0, r3, r0
  ldr r3, .adrpl+4
  str r0, [r3]
  bx  lr
  .align  2
.adrpl:
  .int  .LANCHOR0
  .int  res
```

- ASM-Code simplified

```
.section .data
  .align  2
height:
  .int  5
length:
  .int  6
.section .bss
  .align 2
  .comm res,4
.section .text
mycfn:
  ldr r2, =height  @ load addr of height
  ldr r3, [r2]     @ load height
  ldr r4, =length  @ load addr of length
  ldr r0, [r4]     @ load length
  add r0, r3, r0
  ldr r3, =res     @ load addr of res
  str r0, [r3]     @ store height+length in res
  bx  lr
```

OSTBAYERISCHE TECHNISCHE HOCHSCHULE REGENSBURG

Armarch - memory - details behind „=<label>“ - example simplified further (B)

- Für immediates kann diese noch weiter vereinfacht werden in dem man `ldr <rd>, =<immediate>` verwendet

- ASM-Code simplified

```
.section .data
  .align  2
height:
  .int  0x13435424
length:
  .int  0x13435424
.section .bss
  .align 2
  .comm res,4
.section .text
mycfn:
  ldr r2, =height
  ldr r3, [r2]
  ldr r4, =length
  ldr r0, [r4]
  add r0, r3, r0
  ldr r3, =res
  str r0, [r3]
  bx  lr
```

- ASM-Code simplified further

```
.section .bss
  .align 2
  .comm res,4
.section .text
mycfn:
  ldr r3, =0x13435424 @ load height
  ldr r0, =0x13435424 @ load length
  add r0, r3, r0
  ldr r3, =res     @ load addr of res
  str r0, [r3]     @ store height+
                   @    length in res
  bx  lr
```

Armarch - evolution of ARM architecture

- Acorn Computer entwickelte den ARM1 1985 als upgrade zum 6502 microprocessor
- ARM steht für Acorn RISC Machine
- ARMv4 ist die erste architecture mit vollen 32bit Adressen, und einem dediziertes CPSR
- ARMv4T führte die 16bit thumb instruction ein
- ARMv5TE fügte DSP(fix-point) und floating-point instructions hinzu
- ARMv6 fügte multimedia instructions (SIMD) hinzu und verbesserte den Thumb instruction set
- ARMv7 fügte support für virtualization, security und verbesserte SIMD instructions hinzu
- ARMv8 führte eine komplette neue 64-bit architecture ein

Armarch - evolution of ARM architecture - floating point (B)

- Floating-point ist flexibler als die fixed-point Zahlen der DSP instructions
- Floating-point instructions verwenden zumindest 16 dedizierte 64-bit register (D0-15 für double bzw. S0-31 für single precision)
- Dazu kommt noch das Floating-point Status und Control Register (FPSCR)
- floating-point instructions beginnen mit “v” und haben einen Suffix `.F32` bzw. `.F64`
  (z.B. `VADD.F32, S2, S0, S1`)

Armarch - evolution of ARM architecture - 64bit I/II

- Im Gegensatz zu anderen architectures hat ARM nicht einfach die 32bit register zu 64bit register erweitert
- ARMv8 führt einen neuen instruction set ein und räumt mit ein paar Eigenheiten des alten instruction sets auf
  - zu wenig general purpose register
  - PC in R15 und SP in R13
- ARMv8 instructions sind weiterhin 32bit breit, und sieht dem ARMv7 aber dennoch recht ähnlich
  - 31 64bit register (X0-X30)
  - PC und SP nicht länger Teil der general purpose register
  - X30 ist link register

Armarch - evolution of ARM architecture - 64bit I/II

- ARMv8 instructions sind weiterhin 32bit breit, und sieht dem ARMv7 aber dennoch recht ähnlich (forts.)
  - Data-processing instructions können mit 32bit und 64bit arbeiten
  - loads und stores arbeiten nur mit 64bit
  - Das cond-field wurde für die meisten instructions entfernt (außer branch und einige wenige andere), um Platz für mehr source und destination registers zu schaffen
  - exception handling wurde vereinfacht
  - SIMD registers wurden verdoppelt
  - instructions für AES und SHA wurden hinzugefügt

Armarch - summary (B)

- Die Schlüsselfragen beim Erlernen einer neuen architecture sind:
  - Was ist die data word length?
  - Was sind die register?
  - Wie ist memory organisiert?
  - Was sind die instructions?