HOCHSCHULE ESSLINGEN

LECTURE COMPUTER ARCHITECTURE

ADVANCED MICROPROCESSOR ARCHITECTURES: INTEL X86

RAINER KELLER

ARM CPUs

ARM

Arm-Architecture: spin-off of British computer firm Acorn (BBC-Micro computer)
ARM = **A**corn **R**ISC **M**achine; Ownership then: 43% Acorn, 43% Apple, …
Company produces and sells the Intellectual Property (IP) and takes royalties, i.e.
ARM designs the instruction set, design the cores, the bus and sells licenses.
Design is very flexible (extension, like FP, VFPV, SIMD) and power-efficient…

Licensees (only selection):

SAMSUNG intel BROADCOM AMPERE
RENESAS Qualcomm NXP amazon Alibaba Group 阿里巴巴集团

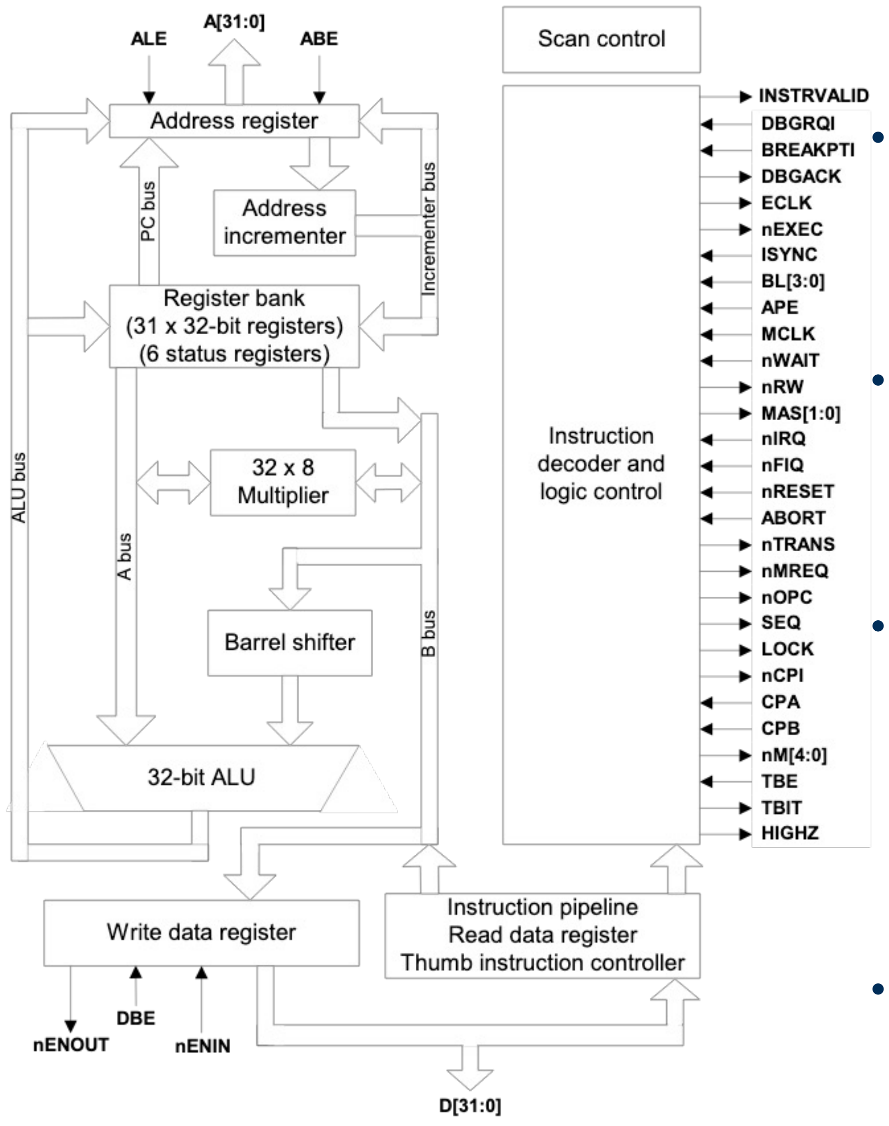
Chips are manufactured by TSMC, UMC, Samsung, GlobalFoundries (in 7-4nm)

Sept. 2020: Nvidia buys ARM for $40 bio!! Nov. 2021: US’ FTC fights takeover

Important:
≤ ARMv7: 32-bit
> ARMv8: 64-bit (released in 2011)

ARM 32-BIT ARCHITECTURE

ARM7:



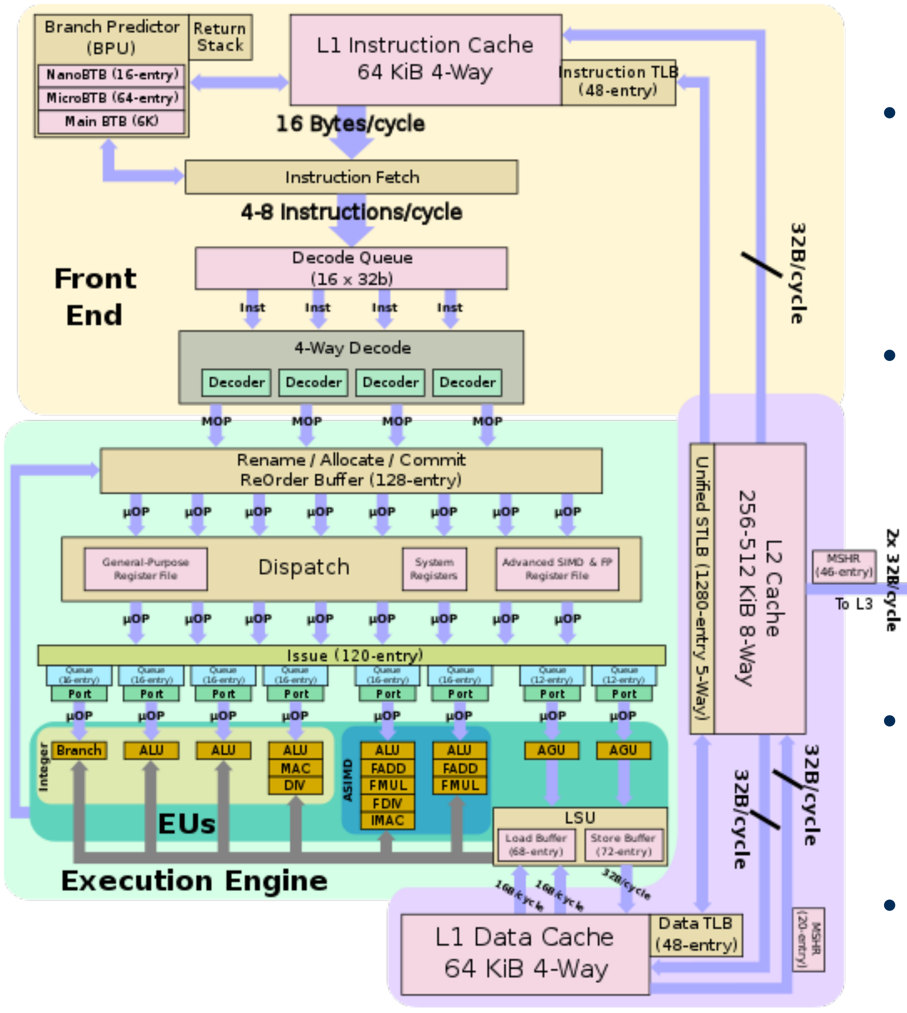
- Typical 32bit RISC architecture (ARM7/9/11, Cortex Mx), extended to 64bit (ARM Ax)
- Family of upwards compatible CPUs from simple microcontrollers without OS to microprocessors for mobile computers
- Tiny version ARM 7 TDMI with von-Neumann-memory interface, no cache, no MPU or MMU, simple integer-ALU, short 3 stage pipeline
- Larger versions ARM 9/11/**Cortex-M**/**Cortex-A** with Harvard memory interface, cache, MPU or MMU in different versions with integer and floating-point ALUs, longer pipelines with 5 to 8 stages, DSP- and Java extensions
- **Increasing market share**, due to:
  High computing performance/Watt, small chip

Quelle: ARM7 Developer Material

ARM 64-BIT ARCHITECTURE

HOCHSCHULE ESSLINGEN

ARM Cortex A-Family: ARMv8 / ARMv9

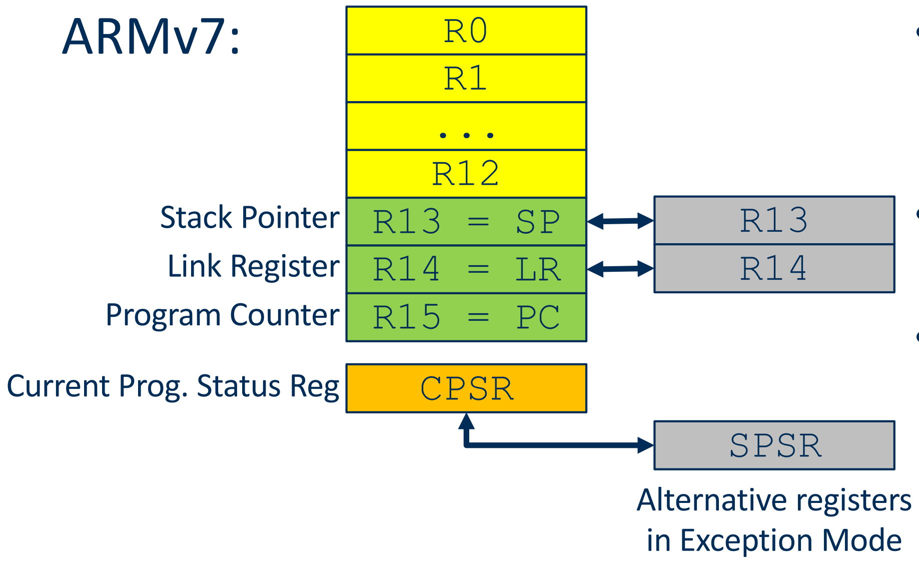


- Target: Server market, attack x86-64
  - → high memory requirements
  - → **Cortex Ax** series
- ISA: **AArch64** with **new 32bit** opcodes, 64-bit addresses and **new register model** with 31 general-purpose 64-bit CPU registers
- Additional AArch32 mode = classical ARM-32bit-architecture, mode switch via software interrupt for backwards compatibility (aka Intel: Now ARM also suffers from its own history!)
- User programs use 32-bit or 64-bit mode, operating systems must run in 64-bit mode
- Multi-core CPUs with superscalar architecture and up to 3 Cache Levels in internal Harvard architecture.

Quelle: ARM8 Developer Material

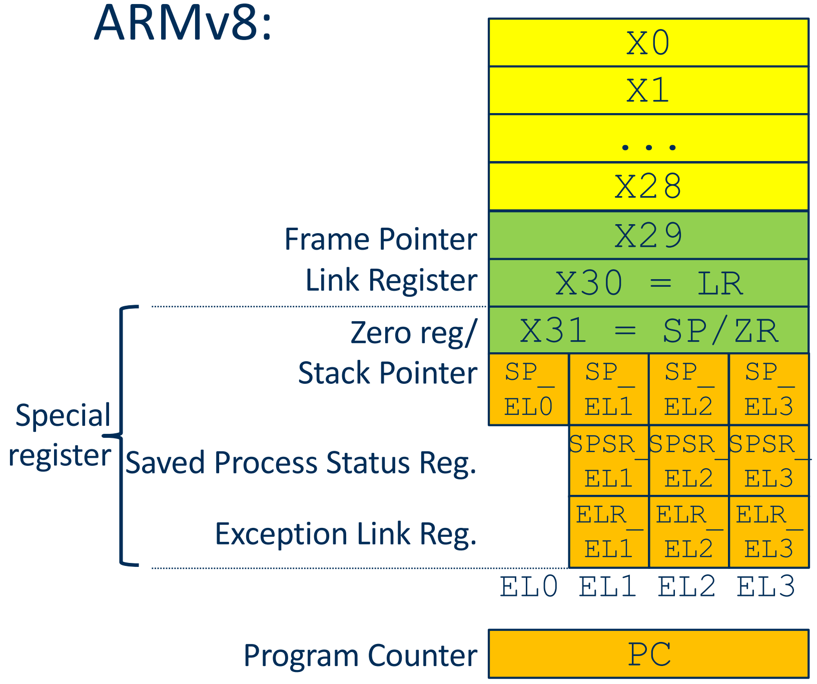
ARM REGISTER SET

ARMv7:



- Register set with 16 registers (incl. 3 special purpose registers (PC, LR, SP)) and a status register (CPSR).
- All registers (incl. PC and SP) can be used as operands in instructions.
- Registers SP and LR will be switched on interrupts (exceptions). After returning from the ISR, the CPU will switch back to the original registers SP and LR again.

ARMv8:



- **31** general purpose registers, each can be used as 64-bit X register (`X0` … `X30`) or as 32-bit W register (`W0` … `W30`): top-most 32-bit are zeroed or sign-extended.

Other registers:

- Separate set of 32 FP (mandatory) and Vector registers (optional for ARMv8!)
- Zero register `XZR` and `WZR` (contains 0, `X31`)
- Stack Pointer (SP) not general purpose (`X31`)
- Program Counter (PC) not general purpose: Read with `ADR Xd, .`
- System registers (ending in `ELx`)

ARM INSTRUCTIONS 1/1

Typical RISC Load- and Store architecture with 3 Address Instructions:

- Originally only ~40 Instructions, still with Aarch64 32-bit → simple instruction decoder.
- Arithmetic-logic instructions use registers and constants only → no memory operands
- Most Arithmetic-logic instruction has 2 operand and 1 result register:
  `ADD R0, R1, R2` ; R0=R1+R2 (Status bits **not** modified)

```
#define u32 uint32_t
uint32_t add (u32 a, u32 b) {
    return a+b;
}
uint64_t minus1 (uint64_t a) {
    return a-1;
}
uint32_t mult(u32 a, u32 b, u32 c) {
    return a*b+c;
}
```

```
             Destination/result reg.
                 Operand 1
                     OP 2 (may include shift)
add:
    ADD W0, W1, W0
    RET
                     May be a constant, too
minus1:
    SUB X0, X0, #1
    RET
         Some instructions have >2 Operands
mult:
    MADD W0, W1, W0, W2
    RET
```

- Each instruction defines if status bits (Negative, Zero, Carry) are set:
  `ADDS R0, R1, R2` ; R0=R1+R2 (Status bits modified)

ARM INSTRUCTIONS 2/2

HOCHSCHULE **ESSLINGEN**

- Most instructions have a 4-bit condition code selector (predicate), to allow for conditional execution (e.g. execute if status bit Zero set). Otherwise it is executed as NOP (**N**o **Op**eration):

```
ADDMI  R0, R1, #4    ; R0 = R1 + 4 (only execute if negative/minus)
ADDMIS R0, R1, #4    ; just the same, but to set the status bits.
```

Instruction Bitmap	No	Condition Code	Executes if	Unsigned	Signed Comparison
0000xxxx …	0	EQ (**Equ**al)	Z		
0001xxxx …	1	NE (**N**ot **E**qual)	~Z		
0010xxxx …	2	CS (**C**arry **S**et)	C	✓	
0011xxxx …	3	CC (**C**arry **C**lear)	~C	✓	
0100xxxx …	4	MI (**Mi**nus)	N		
0101xxxx …	5	PL (**Pl**us)	~N		
0110xxxx …	6	VS (O**v**erflow **S**et)	O		
0111xxxx …	7	VC (O**v**erflow **C**lear)	~O		
1000xxxx …	8	HI (**Hi**gher)	C and ~Z	✓	
1001xxxx …	9	LS (**L**ower or **S**ame)	~C and Z	✓	
1010xxxx …	A	GE (**G**reater or **E**qual)	N = V		✓
1011xxxx …	B	LT (**L**ess **T**han)	N = ~V		✓
1100xxxx …	C	GT (**G**reater **T**han)	(N = V) and ~Z		✓
1101xxxx …	D	LE (**L**ess or **E**qual)	(N = ~V) or Z		✓
1110xxxx …	E	AL (**Al**ways)	True		
1111xxxx …	F	NV (**N**e**v**er)	False		

- Exclusive OR of a condition code with 1 gives opposite condition code.

ARM INSTRUCTIONS 3/3

- The 2nd Operand can be shifted left or right, i.e. multiplied or divided by 2^x, before the instruction is executed. The number of shifts can be specified via a constant or via another register.

  `ADD R0, R1, R2 ASL #4` ; R0 = R1+(R2<<4) $= R1 + R2 \cdot 2^4$

  `ADD R0, R1, R2 ASR R3` ; R0 = R1+(R2>>R3) $= R1 + R2 / 2^{R3}$

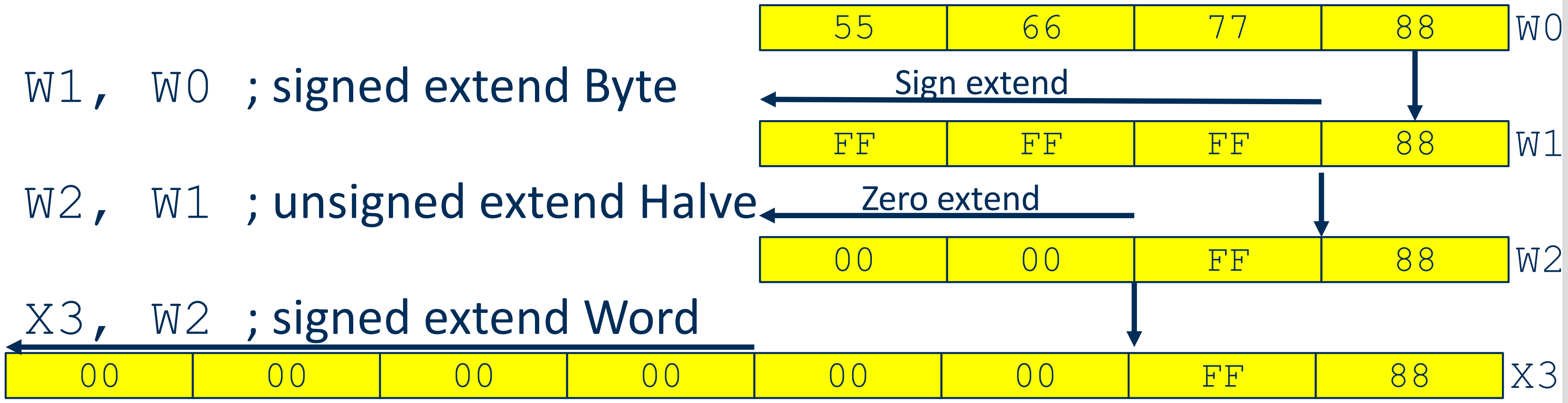
  with `ASL`/`ASR` shifting arithmetically left/right (signed), `LSL`/`LSR` logically shifting left/right (unsigned numbers), `ROR` rotate right.
- Modifying the status bits, bit shifting and conditions are also possible with data transport instructions:

  `MOVEQS R0, R1 ASL #4` ; R0 = R1<<4, if Z bit set, modify status reg.
- In Aarch64, the Instruction may classify operand size using B (Byte), H (Halve, 16-Bit) and W (Word, 32-Bit), i.e. with sign-extend:

  `SXTB W1, W0` ; signed extend Byte

  `UXTH W2, W1` ; unsigned extend Halve

  `SXTW X3, W2` ; signed extend Word



ARM ADDRESSING MODES

Addressing Modes to Load and Store Registers from/to Memory:

```
LDR W0, <source_address>     ; Load 32-bit into W0
LDR X0, <source_address>     ; Load 64-Bit into X0
STRB W0, <dest_address> ; Store 8-Bit  LSB from W0
```

Addressing modes for source and destination `<...address>`:

Direct Address

Register-indirect	`[R0]`	address in R0
	`[R0,#4]`	address is R0 + 4
	`[R0,R1]`	address is R0 + R1
	`[R0,R1,ASL #2]`	address is R0 + (R1<<2)

Pre-Indexed:

In all previous examples the content of `R0` and `R1` will not change. Adding a ! (Register Writeback) to the braces, e.g. `[R0, R1]!`, `R0` will be changed to the newly computed address, i.e. here R0=R0+R1.

Post-Indexed:

With [ ] (without !) around the 1st regi. only, e.g. `[R0],R1,ASL #2`, only the 1st register's content will be used as address, here `R0`, but the address register `R0` nevertheless will be changed to R0=R0+(R1<<2).

ARM32 SUBROUTINE CALLS

- On ARM32 one may operate without stack, if subroutines are not used. A Subroutine call `BL func` (for **B**ranch and **L**ink) stores the return address in register `R14=LR` (ARM32). Aarch64 requires stack.
- When subroutine calls are nested, `R14=LR` is overwritten by the second return address. Therefore, the first return address must be manually saved and restored by the programmer, i.e. a push/pop must be simulated.
- Instructions

  ```
  STR R0,[SP,#-4]!     ; Decrement SP and store R0 on stack
  LDR R0,[SP],#4       ; Read R0 from stack and increment SP
  ```

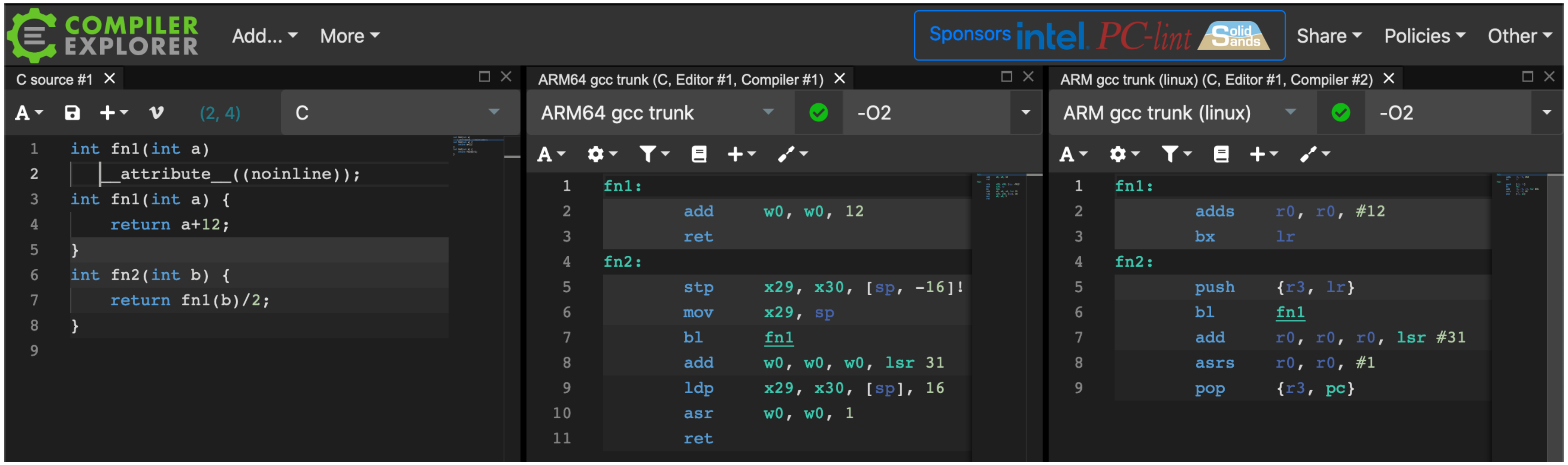
  simulate the well-known PUSH/POP/PULL stack operations of other microprocessors. With

  ```
  STMFD SP!, {R0-R4, R6}
  LDMFD SP!, {R0-R4, R6}
  ```

  A list {...} of registers can be stored/loaded to/from the stack in a single instruction (including LR or PC).

ARM32 VS. AARCH64 COMPILATION

A very good tool for ABI- and Compiler comparison: https://godbolt.org


COMPILER EXPLORER
Add...
More
Sponsors intel PC-lint Solid Sands
Share
Policies
Other
C source #1
(2, 4)
C
ARM64 gcc trunk (C, Editor #1, Compiler #1)
ARM64 gcc trunk
-O2
ARM gcc trunk (linux) (C, Editor #1, Compiler #2)
ARM gcc trunk (linux)
-O2

```c
int fn1(int a)
    __attribute__((noinline));
int fn1(int a) {
    return a+12;
}
int fn2(int b) {
    return fn1(b)/2;
}
```

```
fn1:
        add     w0, w0, 12
        ret
fn2:
        stp     x29, x30, [sp, -16]!
        mov     x29, sp
        bl      fn1
        add     w0, w0, w0, lsr 31
        ldp     x29, x30, [sp], 16
        asr     w0, w0, 1
        ret
```

```
fn1:
        adds    r0, r0, #12
        bx      lr
fn2:
        push    {r3, lr}
        bl      fn1
        add     r0, r0, r0, lsr #31
        asrs    r0, r0, #1
        pop     {r3, pc}
```

Allows selecting different architectures (here Aarch64) vs. ARM (32bit)
Would allow selecting Thumb-Instruction (see next slide)

Allows comparing different compiler vendors (gcc vs. Microsoft)

ARM PROTECTION MODES

- Like 80x86-CPUs ARM-CPUs have privileged and non-privileged operating modes (ARM User Mode, System Mode and various Exception Modes).
- Switching modes is done by setting bits in the status register or automatically by an interrupt or exception. With each mode change register CSPR will be saved and registers LR and SP will be switched, i.e. each mode uses its own stack. Mode switching is a privileged instruction.
- Memory Protection MPU and Memory Management MMU are optional

Advantages/disadvantages of 32bit RISC-instructions vs. 16bit microcontrollers

- 32bit RISC instructions (ARM instruction set) lead to faster, but longer program code com- pared to typical 8/16bit CISC microcontrollers.
- To reduce code size, ARM additionally has a set of 16-bit instructions (Thumb instr. set). Thumb code is ~30% shorter, but 40% slower than 32-bit code.
- Switching between ARM and Thumb is possible when calling subroutines…
- Thumb instructions have several restrictions and are much closer to typical CISC microprocessors:
  - Only 2 rather than 3 register operands per instr. (2-address instruction)
  - No conditional execution of instructions, must use conditional branch
- ARM Cortex M-family, addresses the classical embedded market, only supports Thumb(2) instruction set (no classical 32bit ARM instructions).

APPLE & ARM: HISTORY

Apple uses ARM CPUs since Newton MessagePad 100 (1983!) with ARM6

And again with iPhone (2007)

And for iPod (2010) were producing their own Apple A4 SoC: designed by Apple produces by Samsung:
Arm Cortex A8: 1-core, 45nm, @1 GHz

Late 2022: Apple A16 SoC for iPhone14:
2+4 cores, 5 core GPU, 16 core Neural engine, in 4nm, @3.46 GHz

Macintosh (Desktop) and PowerBook (later called MacBook):

- Until 2006, Apple used IBM Power CPUs…
- From 2006 to 2020, Apple bought Intel U-based CPUs: low-power, high-performance CPUs….
- In 2021: Apple switched to their own "Apple silicon"

HOCHSCHULE ESSLINGEN

LECTURE: LITERATURE

According to list of literature:

- Patterson, D., Hennessy, J.: *Computer Organization and Design*, Kaufmann, 2011
  (Deutsche Übersetzung: Rechnerorganisation und –entwurf, Spektrum)
- Hennessy, J., Patterson, D.: *Computer Architecture: A quantitative Approach, **5th edition***, Morgan Kaufmann, 2017
- Tanenbaum, A., Austin, T.: *Structured Computer Organization*, Pearson, 6th edition, 2013
- Tanenbaum, A., Austin, T.: *Rechnerarchitektur: von der digitalen Logik zum Parallelrechner*, Pearson, 6th ed., 2014
- Huang, H.W.: *The HCS12/9S12. An Introduction to the HW and SW interface*, Thomson Learning, 2009
- Beierlein, T.: *Taschenbuch Mikroprozessortechnik*, Carl-Hanser Verl., 2011