Grundlagen der Informatik

Angewandte Informatik

WS 20/21

Contents

1 Intro

1.1 Representation of numbers characters

Numbers and characters are saved in the memory and need a binary representation. There are different ways how one can represent numbers and characters, depending on the needs the program has. Having a program which needs a counter, only needs positive integers so there is no need for saving decimals. Also the range is important. Is the program counting to 100 or 100 million. Different datatypes need less bytes to store data, but then the range or precision (Genauigkeit) suffers.

Integers

Unsigned (only positive) integers only differ in how many bits they use. Typical sizes are 8-bit (short), 16-bit (half word), 32-bit (word) and 64-bit (double word).
Signed integers need to save the minus symbol somewhere. There are several options to "save the minus". One is just saying if the MSB is 1, the number is negative. The problem is that 0000 and 1000 are both 0, but one is a positive and one is a negative 0 which isn't very effective.
Another implementation is the **one-complement**. Here you just invert every bit to get the "negative version" of the number. Again the ± 0 is possible, but the one-complement creates a symmetry with negative and positive numbers and is needed for the two-complement.
The **two-complement** takes the result from the one complement and adds $+1$ to it. The symmetry is gone but the ± 0 is gone (only positive 0) and an extra negative number is won. It can be calculated like this:

$$Y_z = -z_{N-1} * 2^{N-1} + \sum_{i=0}^{N-2} z_i * 1^i, z_i \in \{0, 1\}$$

One other way to create negative number is by using a **bias/offset**. One needs to define the offset first. Now every number in the memory will be read nad the offset will be subtracted from it. An offset of 128 means that the positive numbers will start at 1000.0000_b[1]. The offset is used in floating point numbers for the exponent.

Decimal numbers

Decimal numbers also have different possible representation. An easy with a fixed point. The number is treated as an integer but at a specific bit, the point is set. The position of the point needs to be defined first. If there are 8-bit to save the number and the point is defined at bit 3, there will be 5 bits for the integer and 3 bits for the mantissa[2]. The problem is, that very big numbers or very small numbers aren't possible.
Floating point numbers fix this by introducing an exponent to the number. The exponent

[1] $1000.0000_b \equiv 0$
[2] Nachkommastellen

has an offset, so it can be negative. A negative exponent makes very small numbers possible, but because the exponent can be positive as well big numbers are possible too. The formula for calculating a normalized float is:

$$f = (-1)^{\text{sign}} \cdot 1.\text{mantissa} \cdot 2^{\text{exponent}}$$

Depending on how many bits the float uses, different values need to be inserted into the formula.

	sign	exponent	mantissa	offset
32-bit	1-bit	8-bit	23-bit	127-bit
64-bit	1-bit	11-bit	52-bit	1023-bit

$$Y = (-1)^s \cdot \left(1 + \sum_{i=-1}^{-F} z_i \cdot 2^i\right) \cdot 2^{e-bias}$$

Besondere Werte wenn der Exponent nur aus 1 besteht. $\pm\infty$ oder NaN (Wenn Mantissa Werte enthält)

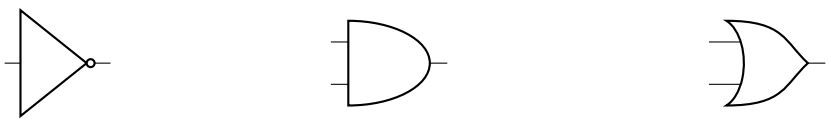
2 Boolean Algebra

Boolean Algebra takes (binary) parameters and return a binary result. There are different operands in boolean algebra:

- NOT $\overline{A}$: takes a single bit and toggles the value (1 -> 0, or 0 -> 1).
- AND $A * B$ or $A \wedge B$: looks if all operands (here just A and B) are set to one and if os returns 1, else returns 0
- OR $A + B$ or $A \vee B$: returns 1 if any operand is set to 1, else all operands need to be 0
- XOR $A \otimes B$: returns one if only 1 parameter is one
- NAND/NOR $\overline{A * B}/A + B$: inverse to AND and OR

With boolean algebra there are some rules:

- DeMorgan's law: $\overline{x} + \overline{y} = \overline{x + y}$ and $\overline{x} + \overline{y} = \overline{x + y}$
- Absorption: $x * (\overline{x} + y) = x * y$
  $x + (\overline{x} * y) = x + y$
  $x * (x + y) = x$
  $x + (x * y) = x$
- Neighborhood: $(x * y) + (\overline{x} + y)$ and $(x + y) * (\overline{x} + y)$



3 Instruction Set Architecture (ISA)

The ISA contains definitions how a processor can be programed. It defines...

- ..the description of instructions (semantics etc)
- ..how the data behaves (how and were the data will be stored and processed)
- ..operation modi (user mode, supervisor mode etc)
- ..and the handling of traps, errors and interrupts

3.1 addresses

addresses are used to store/load data and can be the target of a (un)conditional jump. It is typically divided into 3 categories:

- register storage space: fast, but small and often only a limited number of registers are available
- data storage space: bigger but slower
- instruction storage space: stores the instruction of the ISA

Addressing Modes

- immediate addressing: The instruction receives the data directly (adding a constant to the Accumulator)
- direct addressing: The address for the instruction is hard coded
- register direct addressing: The instruction addresses the register directly (address is constant)
- indirect addressing: first the address is loaded from a specific register and then the memory address is used to processed (register only stores address instead of value)
- indexed indirect addressing: two registers are used to get the address from the value. One is contains the address and one is a counter. Adding both together results in the actual address (arrays)
- program counter relative addressing: like indexed indirect access but the program counter functions as the address counter

3.2 Instructions

Processors work after the control flow principle. The basic idea is that an operation takes operands and generates a result. In order for an processor to run algorithms, the processor needs to be able to process different kinds of operations:

- algorithmic operations: add, subtract, ...
- comparisons: if (greater, lower, equal, ...)
- logic operations: boolean algebra
- shift operations: rotate the byte left/right (multiply/divide by 2)
- control the control-flow: jumps

Instruction can be classified into different types, looking at how many addresses are used. Monadic operations only use one address (NOT for example), dyadic operations use 2 addresses. ADD A,B for example adds A and B together and saves the result in A. Some operations use 0 addresses by addressing implicitly for example the registers or program counter.

3.3 States

An processor need to be able to handle exceptions. Exceptions are differentiated in two groups:

- traps: synchronous events, occur when something in the program happens which shouldn't happen (division by zero)
- interrupts: asynchronous event, occurs when something external from the program needs to be executed (button press, timer)

Operation modes disconnect sensitive areas and none sensitive areas from a computer. A program for viewing pictures doesn't need full access to the whole computer and it's resources. The most basic case is implementing a user-mode which has access to its own files and supervisor mode which can access everything when needed.

4 ISA-ARM

The ARM architecture uses the stored-program concept. Instructions and data are both stored in memory (as numbers). This results into a great bit of flexibility and leads to the stored-program computer.

4.1 Operations

Creating a program typically involves using a programing language instead of assembly languages for convenience reasons. Processors only understand compiled code and depending on the processor the compiled code will look different. Let's say f = (a + b) - (c + d) is c code we want to compile for ARM. ARM only allows arithmetic operations using registers, so first all values of the variables need to be loaded into register. ARM also only allows 2 addresses for adding and subtracting at once so the results need to split into pieces and then stitched together at the end.

4.2 Operands

Operands are in short word (32 bit) and double word (64 bit, size of ARMv8 register size). Registers and variables (from programing languages) are different because registers are limited in size. Too many registers would increase the clock cycle time. Therefore if too many variables were created register values need to be moved into the memory (and vice versa). Those operations are called data transfer instructions.

4.3 Instructions

ARMv8 uses its own assembly language. It's pretty close to the machine code but it still needs to be converted to proper machine code. 'ADD x9, x20, x21' would be translated into '1112 21 0 20 9'. It is divided as following tabling shows (a R Format instruction):

opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

Instruction	Format	opcode
ADD	R	1112_{10}
SUB	R	1624_{10}
ADDI	I	580_{10}
SUBI	I	836_{10}
LDUR	D	1986_{10}
STUR	D	1984_{10}

Example: 1986 240 0 10 9 -> Load the address in register 10 with an offset of 30 bytes (240/8=30) into register 9

- opcode: the numeric representation of the instruction
- Rm: second register
- shamt: shift amount
- Rn: first register
- Rd: destination register

D-Format				
opcode	address	op2	Rn	Rt
11 bits	9 bits	2 bits	5 bits	5 bits

I-Format			
opcode	immediate	Rn	Rd
10 bits	12 bits	5 bits	5 bits

R-Format is often used for arithmetic instructions using addresses or for shifting a register (and saving it to another), while I-Format is often used for immediate instructions. D-Format is used for loading and storing values from/to register to/from memory.

Instruction	ARM code	description
LSL	LSL X11, X19, #4	shifts x19 4 times left and stores result in X11
AND	AND X9, X10, X11	X9 = X10 * X11 (binary AND)
OR	OR X9, X10, X11	X9 = X10 + X11 (binary OR)
EOR	EOR X9, X10, X11	X9 = X10 $\otimes$ X11 (binary EOR/XOR)
NOT	EOR X9, X10, X11(=1111...111)	Not isn't implemented so EOR is used

ANDI, ORRI, EORI are the immediate variations of the above instructions.

Branches

- if: uses 'CBZ Register, label' (jump to 'label' if Register is zero) and CBZN (jump if not zero)
- loop: uses a decreasing counter and CBZ and jumps to the beginning of the loop as long as the counter isn't zero

There are more conditions like less, less or equal, etc.: To check if a branch went out of bounds signed numbers could be treated as unsigned numbers and compared to a negative number (MSB = 1) so out of bounds can be identified.

4.4 Procedures

Procedures are subroutines of a program and are good for implementing abstraction in the program. For a procedures to work the hardware needs to be able to perform the following steps:

1. save parameters in a place where the procedures can access them (X0 - X7)
2. give procedure the control
3. acquire storage resources for the procedure
4. do the task

5. put the result in a place where the main program can access it (X0 - X7)

6. return control the previous procedure (return address saved in LR(X30))

ARMv8 supports the branch-and-link instruction (BL). It branches to the procedure address and writes the return address in X30. This is needed because if a procedure is called by different parts of the program so the return doesn't have to be hardcoded.The caller calculates the return address by adding 4 to the program counter. The current program counter points to the branch so it need to go one step further. The registers should hold the same value after the branch back so registers are saved into a stack before the program branches off. This is called spilling. Here the stack pointer (SP) is needed. Its a register which saves the last spilled address. The operations push and pop, adds or retrieves elements to/from the stack. The stack grows from high to low, so if an element is pushed to the stack, the value in the stack pointer needs to be decreased.
X9 - X17 are registers which aren't preserved by a procedure call, X19 - x28 will be restored if necessary.
C classifies variables into automatic and static, static variables are those declared outside a procedure. In ARMv8 a so called global pointer points to the static area. A lot of ARMv8 compilers reserve X27 as the GP (global pointer).
The stack can be also used to store variables which don't have space in the registers. It a segment in the stack called procedure frame or activation record.

4.5 addresses

Basically nothing to compared to the the already done addressing section.

4.6 program

A programing language like c compiles it's code into assembly code. Assembly code is a symbolic language which will be translated into machine code. It uses the symbol table which matches the names of labels to the corresponding addresses in memory. It creates an object file which typically exists in 6 pieces:

1. file header which describes size and position of the other pieces in the object file

2. text segment which contains the machine language code

3. static data segment (UNIX allows static data which is allocated throughout the program and dynamic data which can grow and shrink as needed)

4. relocation information identifies instructions and data which rely on absolute addresses (and probably adjusts them accordingly)

5. symbol table contains not defined labels like external references

6. debugging information contains descriptions on how a the modules were compiled

The linker or link editor that links the independent compiled modules and resolves the undefined labels to create one executable file. The executable is loaded by the loader (at least in UNIX). It reads the file header to determine the size of the text and data segments. Then creates an address space large enough to fit all and copies the data into memory. If there are parameters they will also be copied into memory. The registers of the processor are initialized and the stack pointer is set to the first free location. At last it branches to a start up routine which initializes the argument registers with parameters and then calls the main routine. If the main routine is done the program terminates with an 'exit' system call.

5 Peripherals

A computer system needs to be able to support different kinds of peripherals.

5.1 Bus System

Bus-Systems in computers are there so different components are able to talk to each other. Typically the critical components (cpu, ram,...) have their own bus system (AHP or ASP bus system) and peripherals or none critical components (keyboard, interrupt time,...) share another bus system (APB bus system). They are connected via so called 'bridge', which connects (bridges) both bus systems if needed (if the keyboard input needs to be processed by the cpu).

5.2 Interrupt, Exceptions, Trap

Sometimes the flow of a program needs to be interrupted, for example when a button is pressed. A program could see in a while loop if a button was pressed and react accordingly, but this wastes resources and as long as the while loop is active, the program can't do anything else. Because a program is deterministic (if everything stays the same, the program will always behave in the same way), it can be interrupted. If the interrupt is done (interrupt function is done) the pre interrupt state needs to be restored so the program can work as it did before.
What an interrupt is and what an exceptions is defined by the processor manufacturer. Interrupts and exceptions can occur internal or external though. Internal means that an invalid instructions is requested (user wants to access admin resources, div by zero) and external typically means that an the program is influenced from outside of the computer (usb stick is plugged in, power button is pressed).
If an exception occurs, a branch will happen. The address of the branch target can either be fixed (division by zero interrupt has an own constant address, and other exceptions also have constant addresses) or the branch address is saved in a exception table where the processor first reads the address (depending on the exception) and then jumps to the given address. Context switches (when registers need to be saved so another (part of a) program can be executed):

1. process change

2. exceptions
3. sub-program calls

6 Bus Systems

There are many different kinds off bus systems and they can be classified in multiple categories:

- on/off chip busses
- parallel/serial busses
- synchronous/asynchronous busses
- automotive busses
- (and more)

In general, busses consists of a couple of wires and are designed to allow computer components to communicate to each other. Parallel busses couple at least two wires to transfer information in (nearly) the same direction (example: cpu uses 5 wires to write to a disc). There are often multiple data sources (sender) and data sink (receiver) on a bus (a sink can also be a source and vice versa). Other problems are that it needs to be defined how data can travel from source to sink so there also must be a well defined timing behavior.

6.1 General Definitions

See slide in moodle, too compact to summarize here.

6.2 UART, RS232

Before the data can be send, sender and receiver need to agree on some parameters of the transfer:

- full or half duplex
- number of bts per character
- band rate speed
- use parity or not
- if parity is used, how many bits
- number of stop bits (at least the number receiver needs)
- mark and space symbols

8N1 was a common implementation. 8 data bits, one stop bit and no parity bit (+ 1 start bit). If the band rate is set to then, then one just needs divide the signal rate by ten to get how many character were sent per second.

6.3 I^2C

I^2C(also called 'I two C') is a multi master bus and is commonly used for low speed peripherals. Multi master means that any component on the bus can be a master and communicate with the other components on the bus (they are called slaves). Masters can also switch as soon the current master is finished with its operations. SMBbus is a subset of I^2C and defines stricter protocols and electrical conventions to promote robustness and interoperability. A master is the component who issues the clock for the timing while the slave receives the clock.
A master is initially in master transmit mode. It sends a start bit, followed by the salve address the master wants to communicate followed by one bit which represents if the master wants to read(1) or write (0). If the slave exists it sends an ACK (acknowledge) bit and as soon the master receives it, the master changes it mode to transfer/receive (depending what the master wants to do) and the slave changes to its complementary mode.
I^2C defines three basic messaging types, while each begin with START and end with a STPO:

1. single message where master writes to slave
2. single message where master reads from slave
3. 'combined' message where master reads or writes at least 2 times to one or more slaves

7 miscellaneous

7.1 ASCII table

Dez	Hex	Okt	Zeichen	Dez	Hex	Okt	Zeichen
0	0x00	000	NUL	32	0x20	040	SP
1	0x01	001	SOH	33	0x21	041	!
2	0x02	002	STX	34	0x22	042	"'
3	0x03	003	ETX	35	0x23	043	#
4	0x04	004	EOT	36	0x24	044	$
5	0x05	005	ENQ	37	0x25	045	%
6	0x06	006	ACK	38	0x26	046	&
7	0x07	007	BEL	39	0x27	047	'
8	0x08	010	BS	40	0x28	050	(
9	0x09	011	TAB	41	0x29	051	)
10	0x0A	012	LF	42	0x2A	052	*

11	0x0B	013	VT	43	0x2B	053	+
12	0x0C	014	FF	44	0x2C	054	,
13	0x0D	015	CR	45	0x2D	055	-
14	0x0E	016	SO	46	0x2E	056	.
15	0x0F	017	SI	47	0x2F	057	/
16	0x10	020	DLE	48	0x30	060	0
17	0x11	021	DC1	49	0x31	061	1
18	0x12	022	DC2	50	0x32	062	2
19	0x13	023	DC3	51	0x33	063	3
20	0x14	024	DC4	52	0x34	064	4
21	0x15	025	NAK	53	0x35	065	5
22	0x16	026	SYN	54	0x36	066	6
23	0x17	027	ETB	55	0x37	067	7
24	0x18	030	CAN	56	0x38	070	8
25	0x19	031	EM	57	0x39	071	9
26	0x1A	032	SUB	58	0x3A	072	:
27	0x1B	033	ESC	59	0x3B	073	;
28	0x1C	034	FS	60	0x3C	074	"<
29	0x1D	035	GS	61	0x3D	075	=
30	0x1E	036	RS	62	0x3E	076	">
31	0x1F	037	US	63	0x3F	077	?

Dez	Hex	Okt	Zeichen	Dez	Hex	Okt	Zeichen
64	0x40	100	@	96	0x60	140	‘
65	0x41	101	A	97	0x61	141	a
66	0x42	102	B	98	0x62	142	b
67	0x43	103	C	99	0x63	143	c
68	0x44	104	D	100	0x64	144	d
69	0x45	105	E	101	0x65	145	e
70	0x46	106	F	102	0x66	146	f
71	0x47	107	G	103	0x67	147	g
72	0x48	110	H	104	0x68	150	h
73	0x49	111	I	105	0x69	151	i
74	0x4A	112	J	106	0x6A	152	j
75	0x4B	113	K	107	0x6B	153	k
76	0x4C	114	L	108	0x6C	154	l
77	0x4D	115	M	109	0x6D	155	m
78	0x4E	116	N	110	0x6E	156	n
79	0x4F	117	O	111	0x6F	157	o
80	0x50	120	P	112	0x70	160	p
81	0x51	121	Q	113	0x71	161	q
82	0x52	122	R	114	0x72	162	r
83	0x53	123	S	115	0x73	163	s

84	0x54	124	T	116	0x74	164	t
85	0x55	125	U	117	0x75	165	u
86	0x56	126	V	118	0x76	166	v
87	0x57	127	W	119	0x77	167	w
88	0x58	130	X	120	0x78	170	x
89	0x59	131	Y	121	0x79	171	y
90	0x5A	132	Z	122	0x7A	172	z
91	0x5B	133	[	123	0x7B	173	{
92	0x5C	134	\	124	0x7C	174	|
93	0x5D	135	]	125	0x7D	175	}
94	0x5E	136	^	126	0x7E	176	"
95	0x5F	137	_	127	0x7F	177	DEL