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 charachters, depending on the needs the program has. Having a programm which needs a counter, only nedds positive integers so there is no need for saving decimals. Also the range is important. Is the programm counting to 100 or 100 milion. Different datatypes need less bytes to store data, but then the range or (Genauigketi) suffers.

Integers

With unsigned (only positive) integers only differ in how many bits they use. Typicall 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 effictive.
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 symmetrie with negative and ppostive numbers and is needed for the two-complement. The two-complement takes the result from the one complement and adds $+1$ to it. The symmetrie is gone but the ± 0 is gone (only positve 0) and an extra negative numebr is won.
One other way to create negative nummber 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 postive 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 definded 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 probelm 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 exponetn has an offset, so it can be negativ. A neagtive exponent makes very small numbers possible, but because the exponent can be positive aswell big numbers are possible too. The formular fot calculating a normalized flaot is:

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

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

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

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

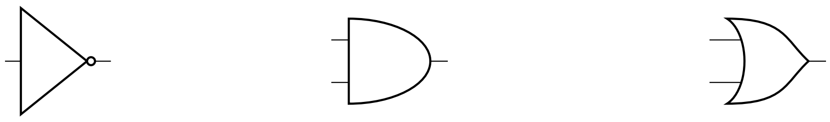
2 Boolean Algebra

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 $\overline{x} + \overline{y} = \overline{x + y}$
- Absorbtion: $x * (\overline{x} + y) = x * y$
  $x + (\overline{x} * y) = x + y$
  $x * (x + y) = x$
  $x + (x * y) = x$
- Neighbourhood: $(x * y) + (\overline{x} + y)$ and $(x + y) * (\overline{x} + y)$



3 Instruction Set Archetecture (ISA)

The ISA contains defintions how a processor can be programmed. It defines...

- ..the descrtiption of instructions (semantics etc)
- ..how the data beahves (how and were the data will be stored and processed)
- ..operation modi (usermod, supervisor mode etc)
- ..and the handling of traps, errors and interupts

3.1 Adreesses

Adresses 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 avaiable
- data storage space: bigger but slower
- instruction storage space: stores the instruction of the ISA

Adressing Modes

- immedeiate adressing: The instruction recieves the data directly (adding a constant to teh Accumulator)
- direct adressing: The adress for the instruction is hard coded ()
- register direct adressing: The instruction adresses the register directly (adress is constant)
- indirect adressing: first the adress is laoded form a specific register and then the memory address is used to processed (register only stores adress instead of value)
- indexed indirect adressing: two registers are used to get the adress form the value. One is contains the adress and one is a counter. Adding both together resluts in the actual adress (arrays)
- programm counter realtive adressing: like indexed indirect access but the programm counter functions as the adress counter

3.2 Instructions

Processors work after the control flow principle. The basic idea is that an operations 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, ...
- comaprisons: 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 differnent types, looking at how many adresses are used. Monadic operations only use one adress (NOT for example), dyadic operations use 2 adresses. ADD A,B for example adds A and B together and saves the result in A. Some operations use 0 adresses by adressing implicitly for example the registers or programm counter.

3.3 States

An processor need to be able to handle expceptions. Expceptions are differentiated in two groups:

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

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

4 ISA-ARM

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

4.1 Operations

Creating a programm typically involves using a programming language instead of assembly langeues for conveniance 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 awant 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 adresses for adding and subtracting at once so the results need to split into pieces and then stiched 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 programming 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 isntruction):

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

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

D-Format				
opcode	adress	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 adresses or for shifting a register (and savin 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	descrtiption
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 nummbers 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 implementating abstracion 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 contorl
3. acquire storage resources for the procedure
4. do the task
5. put the result in a place where the main programm can access it (X0 - X7)
6. return control the previous procedure (return adress saved in LR(X30))

ARMv8 supports the branch-and-link instruction (BL). It branches to the procedure adress and writes the return adress in X30. This is needed because if a procedure is called by different parts of the programm so the return doesn't have to be hardcoded.The caller callculates the return adress by adding 4 to the programm counter. The current programm counter points to the branch so it need to go one step fourther. 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 adress. 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 calle, X19 - x28 will be restored if neccesarray.
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 acalled procedure frame or activation record.

4.5 Adresses

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

4.6 Programm

A programming 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 naems of labels to the corresponding adresses in memory. It creates an object file which typically exsist 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 programm and dynamic data which can grow and shrink as needed)
4. relocation information indentifies instructiosn and data which rely on absolute adresses (and probably adjusts them accordingly)
5. symbol table contains not defined labels like external references
6. debuging information containsa 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 adress space large enough to fit all and copies the data into memory. If there are parameters thy will also be copied into memory. The registers of the processor are initilized and the stack pointer is set to the first free location. At last it branches to a start up routine which initilizes the argumetn registers with parameters and then calls the main routine. If the main routine is done the programm terminates with an 'exit' system call.

5 miscelanious

5.1 ASCII tabel

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	ˆ	126	0x7E	176	"
95	0x5F	137	_	127	0x7F	177	DEL