Digitaltechnik Zusammenfassung 2009 [1]

Revision 26

Stefan Heule

12. Oktober 2009

[1] Licence: Creative Commons Attribution-Share Alike 3.0 Unported (http://creativecommons.org/licenses/by-sa/3.0/)

1 Kombinatorische Schaltungen

1.1 Gatter

	Logik	Algebra	Schaltbild	Verilog
OR	$x \vee y$	$x + y$		`x || y`
NOR	$\neg(x \vee y)$	$\overline{x + y}$		
AND	$x \wedge y$	$x \cdot y$		`x && y`
NAND	$\neg(x \wedge y)$	$\overline{x \cdot y}$		
NOT	$\neg x$	$\overline{x}$		`!x`
XOR	$x \neq y$	$x \oplus y$		`x ^ y`
XNOR	$x = y$	$\overline{x \oplus y}$		`x == y`
	$x \rightarrow y$	$x \leq y$		`!x || y`

1.2 Wahrheitstabellen

a	b	$\neg a$	$a \wedge b$	$a \vee b$	$a \rightarrow b$	$a \leftrightarrow b$	$a \oplus b$
1	1	0	1	1	1	1	0
1	0	0	0	1	0	0	1
0	1	1	0	1	1	0	1
0	0	1	0	0	1	1	0

1.3 Normalformen

- **DNF**: Disjunktive Normalform. In der Wahrheitstabelle können die "1-Zeilen" verodert werden.
  ($0 \rightarrow \neg x, 1 \rightarrow x$)
  - **Literal**: negierte oder unnegierte Variable
  - **Produktterm**: Konjunktion von Literalen
  - **Minterm**: maximaler Produktterm (enthält alle Variablen)
- **KNF**: Konjunktive Normalform. In der Wahrheitstabelle können die "0-Zeilen" verundet werden.
  ($0 \rightarrow x, 1 \rightarrow \neg x$)

1.4 Umformungen

$$\begin{aligned} a \oplus b &= (a \wedge \neg b) \vee (\neg a \wedge b) && = a\overline{b} + \overline{a}b \\ &= (a \vee b) \wedge (\neg a \vee \neg b) && = (a + b)(\overline{a} + \overline{b}) \\ &= (a \vee b) \wedge \neg(a \wedge b) && = (a + b)\overline{(ab)} \end{aligned}$$

1.5 Frequenzberechnung

$$\text{Zykluszeit:} \quad t_z = t_s + t_p + t_l$$

$$\text{Max. Frequenz:} \quad f_{max} = \frac{1}{t_z} = \frac{1}{t_s + t_p + t_l}$$

t_s : Setupzeit

t_p : Propagierungsdelay der Flipflops

t_l : Längster Pfad

1.6 Rechenregeln

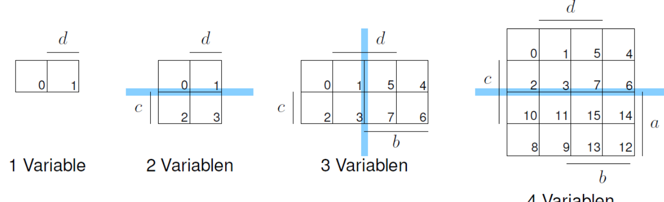
Kommutativität	$x \wedge y \equiv y \wedge y$ $x \vee y \equiv y \vee x$
Assoziativität	$x \wedge (y \wedge z) \equiv (x \wedge y) \wedge z$ $x \vee (y \vee z) \equiv (x \vee y) \vee z$
Distributivität	$x \wedge (y \vee z) \equiv (x \wedge y) \vee (x \wedge z)$ $x \vee (y \wedge z) \equiv (x \vee y) \wedge (x \vee z)$
De'Morgan	$\neg(x \wedge y) \equiv \neg x \vee \neg y$ $\neg(x \vee y) \equiv \neg x \wedge \neg y$
Idempotenz	$x \wedge x \equiv x$ $x \vee x \equiv x$
Controlling Value	$x \wedge 0 \equiv 0$ $x \vee 1 \equiv 1$
Neutraler Wert	$x \wedge 1 \equiv x$ $x \vee 0 \equiv x$
Doppelte Negation	$\neg(\neg x) \equiv x$
Abschwächung	$x \wedge y \equiv x$ wenn $x \Rightarrow y$ y schwächer als x
Verstärkung	$x \vee y \equiv x$ wenn $y \Rightarrow x$ y stärker als x
Konsensus	$x \cdot y + \overline{x} \cdot z + y \cdot z \equiv x \cdot y + \overline{x} \cdot z$
Shannon-Expansion	$e \equiv x \wedge e[1/x] \vee \neg x \wedge e[0/x]$

1.7 Bindung der Operatoren

stärker $\quad \neg \quad \wedge \quad \vee \quad \rightarrow \quad \oplus \quad \leftrightarrow \quad$ schwächer

2 Formel-Minimierung

2.1 Karnaugh-Maps



2.2 Quine-McCluskey

2.2.1 Definitionen

- Ein **Implikant** einer DNF ist ein implizierender Produktterm, z.B. ein Monom oder der Konsens zweier Monome. Entspricht einem Block im Karnaugh-Diagramm

- Ein **Primimplikant** ist ein maximaler Implikant und entsprich einem maximalen Block, der in keine Richtung erweitert werden kann.

- Ein **Kernimplikant** überdeckt einen sonst nicht überdeckten Minterm.

- Ein **redundanter Primimplikant** dagegen wird von einem Kernimplikanten überdeckt.

2.2.2 Vorgehen

- Alle Wahrheitsbelegungen mit Wahrheitswert 1 aus der Funktionstabelle auswählen.

- Mit der einfachen Konsensusregel alle Primimplikanten bilden.
  Dazu werden die Formeln in eine Tabelle geschrieben und nach Anzahl Einsen sortiert. Nun können jeweils benachbarte Bereiche verglichen werden und wenn sich zwei Formeln in genau einem Bit unterscheiden, darf dort ein "Don't care" eingefügt werden.

- Kernimplikanten suchen, redundante Primimplikanten entfernen und Überdeckung aller Primimplikanten bestimmen.

- Die Disjunktion der Primimplikanten der Überdeckung ergibt die gewünschte, minimierte Formel.

2.2.3 Beispiel

Minterm Indizes	a	b	c	d	Anzahl Einsen	Primimplikant
0	0	0	0	0	0	nein
2	0	0	1	0	1	nein
8	1	0	0	0	1	nein
5	0	1	0	1	2	nein
10	1	0	1	0	2	nein
12	1	1	0	0	2	nein
13	1	1	0	1	3	nein
15	1	1	1	1	4	nein

	0	2	5	8	10	12	13	15
8,12				⊠		×		
5,13			⊗				⊗	
12,13						×	⊠	
13,15							⊗	⊗
0,2,8,10	⊗	⊗		⊗	⊗			

× = Minterme in Primimplikanten
⊗ = Minterme in Kernimplikanten
⊠ = Minterme überdeckt von Kernimplikanten

3 Assembler

3.1 Register

Index	0	1	2	3	4	5	6	7
	000	001	010	011	100	101	110	111
Name	`eax`	`ecx`	`edx`	`ebx`	`esp`	`ebp`	`esi`	`edi`

3.2 Byte-Order

Y86 arbeitet im Little-Endian Format. Es wird also das Byte mit den niederwertigsten Bits an der kleinsten Speicheradresse gespeichert.

Beispiel:	`0x 1A 2B 3C 4D` = 439041101
Little-Endian:	`0x 1A 2B 3C 4D`
Big-Endian:	`0x 4D 3C 2B 1A`

3.3 Speicherzugriffe

Speicherzugriffe sind nur mit dem `esi`-Register möglich:

$$\texttt{ea} = \texttt{esi} + \text{Displacement}$$

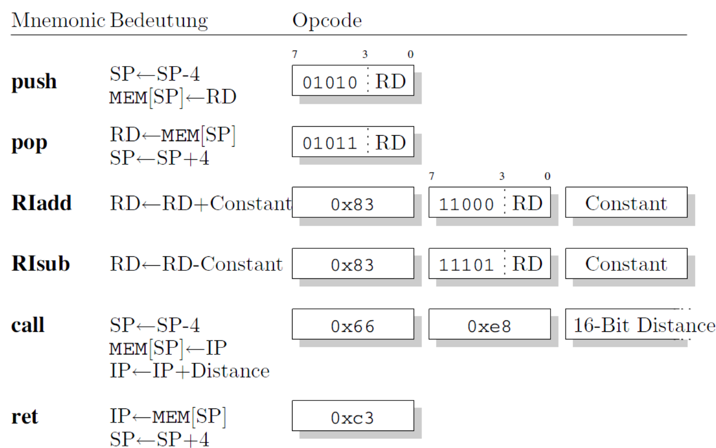
3.4 Binärzahlen

0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	a
3	0011	3	11	1011	b
4	0100	4	12	1100	c
5	0101	5	13	1101	d
6	0110	6	14	1110	e
7	0111	7	15	1111	f

$2^0 = 1$	$2^8 = 256$
$2^1 = 2$	$2^9 = 512$
$2^2 = 4$	$2^{10} = 1024$
$2^3 = 8$	$2^{11} = 2048$
$2^4 = 16$	$2^{12} = 4096$
$2^5 = 32$	$2^{13} = 8192$
$2^6 = 64$	$2^{14} = 16384$
$2^7 = 128$	$2^{15} = 32768$

3.5 Befehle

Mnemonic	Bedeutung	Opcode		
			7 6 3 0	
add	RD←RD+RS	`01`	`11` RS RD	
sub	RD←RD-RS	`29`	`11` RS RD	
jnz	`if`(¬ZF) IP←IP+Distance	`75`	Distance	
RRmov	RD←RS	`89`	`11` RS RD	
RMmov	`MEM[ea]`←RS	`89`	`01` RS `110`	Displacement
MRmov	RS←`MEM[ea]`	`8b`	`01` RS `110`	Displacement
hlt		`f4`		



Mnemonic	Bedeutung	Opcode		
push	SP←SP-4 `MEM`[SP]←RD	`01010` RD		
pop	RD←`MEM`[SP] SP←SP+4	`01011` RD		
RIadd	RD←RD+Constant	`0x83`	`11000` RD	Constant
RIsub	RD←RD-Constant	`0x83`	`11101` RD	Constant
call	SP←SP-4 `MEM`[SP]←IP IP←IP+Distance	`0x66`	`0xe8`	16-Bit Distance
ret	IP←`MEM`[SP] SP←SP+4	`0xc3`		

3.6 Beispiele

3.6.1 Schleife, $i = 1, \ldots, 10$

```
Intitalisieren:
    sub esi, esi
    sub eax, eax
    sub ebx, ebx
Anfang:
    ;; Speichere 1 in edx
l   mov edx, [BYTE one+esi]
    add eax, edx
Berechnung:
    add ebx, eax
Ende:
    mov ecx, eax
    mov edx, [BYTE ten+esi]
    sub ecx, edx
    jnz l
    hlt
one dd 1
ten dd 10
```

3.6.2 if-then-else

```
    ; if (a == b) then
    mov eax, [BYTE a+esi ]
    mov ebx, [BYTE b+esi ]
    sub eax, ebx
    jnz f
    ; Code fuer 'then'
    mov eax, [BYTE one+esi ]
    add eax, eax
    jnz e
f:  ; Code fuer 'else'
e:  ; ...
```

3.6.3 Array invertieren

```
    sub esi, esi
    mov edi, [BYTE four + esi]
    ; eax = 0
    sub eax, eax
    ; ebx = len
    mov ebx, [BYTE len + esi]
    ; ecx = len
    mov ebx, [BYTE len + esi]
L1:
    ; edx = [eax + data]
    mov esi, eax
    mov edx, [BYTE data + esi]

    ; ebp = [--ebx + data]
    sub ebx, edi
    mov esi, ebx
    mov ebp, [BYTE data + esi]
    ; [ebx + data] = edx
    mov [BYTE data + esi], edx
    mov esi, eax
    ; [eax + data] = ebp
    mov [BYTE data + esi], ebp
    ; eax += 4
    add eax, edi
    ; ecx -= 8
    sub ecx, edi
    sub ecx, edi
    jnz L1

    hlt
four:   dd 4
len:    dd 16
data:   ; ...
```

3.6.4 Array kopieren

```
    sub esi, esi
    mov edi, [BYTE four + esi]
    mov eax, [BYTE a + esi]
    mov ebx, [BYTE b + esi]
    mov ecx, [BYTE c + esi]
L1:
    mov esi, eax
    mov edx, [esi]
    add eax, edi
    mov esi, ebx
    mov [esi], edx
    add ebx, edi
    sub ecx, edi
    jnz L1
; ...
four:   dd 4      ; Bytes einer Zahl
a:      dd 1234   ; Quellarray
b:      dd 5678   ; Zielarray
c:      dd 20     ; Laenge in Bytes
        ; c % 4 = 0 && c != 0
        ; b <= a || b >= a+c
```

4 Verilog

4.1 Operatoren

<table>
<tr><td>+, -, *</td><td>Arithmetic operators</td></tr>
<tr><td>/</td><td>Integer division (fractional part truncated)</td></tr>
<tr><td>%</td><td>Modulo (takes sign of the first operand)</td></tr>
<tr><td>**</td><td>Exponent</td></tr>
<tr><td>-</td><td>Negation (2's complement)</td></tr>
<tr><td>~</td><td>Bitwise NOT (1's complement)</td></tr>
<tr><td>&</td><td>Bitwise AND</td></tr>
<tr><td>|</td><td>Bitwise OR</td></tr>
<tr><td>^</td><td>Bitwise XOR</td></tr>
<tr><td>^~, ~^</td><td>Bitwise XNOR</td></tr>
<tr><td><<, >></td><td>Logical shift (padding with 0)</td></tr>
<tr><td><<<, >>></td><td>Arithmetic shift (padding with leftmost bit when shifting right)</td></tr>
<tr><td>?:</td><td>Conditional</td></tr>
<tr><td>{ a, b }</td><td>Concatenation</td></tr>
<tr><td>{ num { a }}</td><td>Replication</td></tr>
</table>

4.2 Vergleiche

>, <, >=, <=	Relational operators (0, 1 or x)
==, ==	Logical equality (0, 1 or x)
===, !==	Case equality (0 or 1)

4.3 Konstanten

Angabe mit [<width in bits>]'<base><number>

- 'b (binär, 2)
- 'o (oktal, 8)
- 'd (dezimal, 10)
- 'h (hexadezimal, 16)

z.B. -4'd3 (1101), 4'b11 (0011), 'h08FF (16 bit hex)

4.4 Module

In Verilog bestehen die Modelle aus Modulen, wobei jedes Modul ein Interface besitzt, welches die Inputs und Outputs definiert. Ein Modul kann dann instanziert werden, wobei ihm ein eindeutiger Name zugewiesen wird.

```
// modul definieren
module abc(input a,b, input [3:0] data, output out);
    wire x;
    assign x = (!a || b) ^ a;
    assign out = x ;
endmodule

// module instanzieren
module main(..)
    abc A1(a,x,data,myvar);
endmodule
```

4.5 always

Mit always lassen sich Endlosschleifen ausdrücken. Zusätzlich kann die Ausführung auf positive/negative Flanken einer bestimmten Variable eingeschränkt werden.

```
always @([posedge | negedge] var) begin .. end
```

Zeitauflösung/Delays

```
'timescale <Zeiteinheit>/<Aufloesung>
```

Wird im Header eines Moduls angegeben. z.B. 1ns/100ps bedeutet dass der Delay-Operator #1 für 1ns verzögert und die Simulation in 100ps Zeitschritten läuft.

Zuweisungsoperatoren

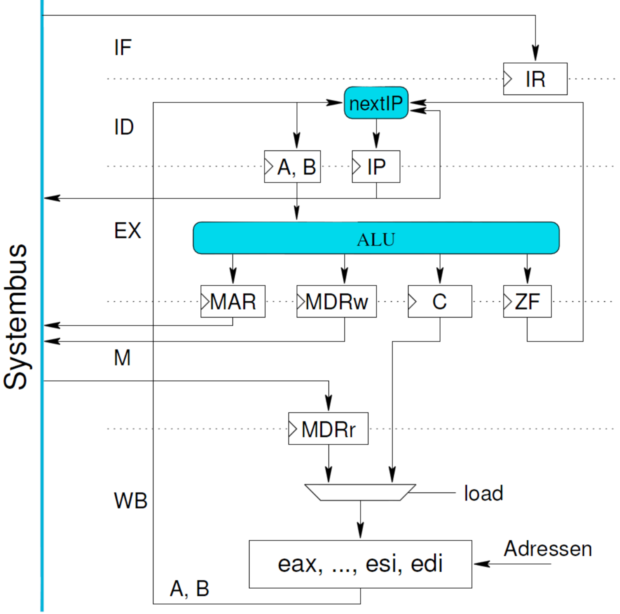
a = b	*blocking assignment*: Der Wert der Zielvariable wird sofort aktualisiert.
a <= b	*non-blocking assignment*: Die rechte Seite wird sofort ausgewertet, die Zuweisung erfolgt jedoch erst im nächsten Zeitschritt (nicht Flanke/Takt!), was eine parallele Ausführung solcher Befehle ermöglicht. Die NBA's bewirken keine Verzögerung im Timingdiagramm (ausser wenn die Auflösung gleich der Zeiteinheit ist, was nicht sehr sinnvoll ist).
assign a = b	*continuous assignment*: Weist der Zielvariable sofort den Wert des Ausdrucks zu, wenn immer dieser ändert. Ausserhalb von always und initial Blöcken!

Weitere Konstrukte

reg a,b;	Lokale 1-Bit Variablen (Register)
reg [3:0] d;	Lokale 4-Bit Variablen
wire a	Draht, verbindet z.B. Ein-/Ausgänge von Modulen
assign #3 a = b & c	Zuweisung (Delay: 3 Einheiten)

5 Der Y86-Prozessor

5.1 Schema



5.2 Vergleiche

CF: Carry-Flag (c_n bei Addition und $\neg c_n$ sonst)
SF: Sign-Flag (s_{n-1})
OF: Overflow-Flag ($c_n \oplus c_{n-1}$)

Befehle	Flags
jz, je	**ZF**
jnz, jne	¬**ZF**
jnae, jb	**CF**
jae, jnb	¬**CF**
jna, jbe	**CF** ∨ **ZF**
ja, jnbe	¬(**CF** ∨ **ZF**)
jnge, jl	**SF** ⊕ **OF**
jge, jnl	¬(**SF** ⊕ **OF**)
jng, jle	((**SF** ⊕ **OF**) ∨ **ZF**)
jg, jnle	¬((**SF** ⊕ **OF**) ∨ **ZF**)
jmp	unbedingt

g	greater
l	less
a	above
b	below
n	not
z	zero

Welcher Befehl eingesetzt wird, hängt davon ab, ob die Zahl ein Vorzeichen hat oder nicht:
mit Vorzeichen: je, jg, jge, jl, jle
ohne Vorzeichen: je, ja, jae, jb, jbe

5.3 Phasen der CPU

Instruction Fetch (IF) Lädt das Instruktionswort aus dem RAM in das Register IR

Instruction Decode (ID) Lädt die Operanden aus dem Register File in die Register A und B, Inkrementierung des IPs

Execute (EX) Ausführung einer evtl. add/sub Instruktion. Addressarithmetik für RAM-Zugriffe

Memory (M) Zugriff auf das RAM

Write-Back (WB) Speicherung des Ergebnisses von add /sub und MRmov im Register File

5.4 Instruktion

IR : | 0 1 2 3 4 5 6 7 (opcode) | 8 9 10 (RD) | 11 12 13 (RS) | 14 15 (mod) | 16 ... 23 (displacement) |

5.5 Register

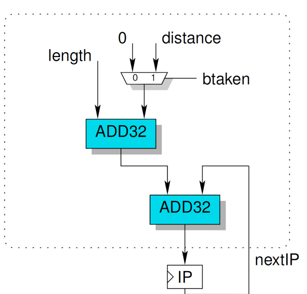
MAR Memory Address Register, speichert ea

C Ergebnis der ALU (arithmetisches Ergebnis)

MDRw Ergebnis der ALU (Adresse)

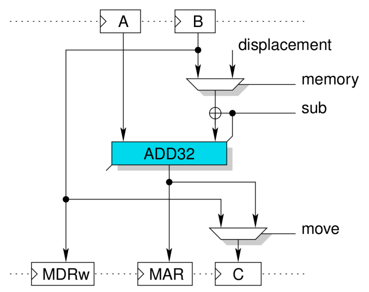
5.6 NextIP Berechnung

```
wire btaken=jnz && ! ZF;

wire [1:0] length=memory?3:(aluop || move || jnz)?2:1;

always @(posedge clk)
    if (ue[1]) begin
        IP = IP+length
        if (btaken) IP = IP+distance;
    end
end
```



5.7 ALU (Arithmetic-Logic-Unit)

```
wire [31:0] ALU_op2=memory?displacement:sub?~B:B;
// Falls memory == 1
//  ALU_op2 = displacement
// Sonst
//      Falls (sub=1): ALU_op2 = B negiert
//      Sonst: ALU_op2 = B
//  End
wire [31:0] ALUout=A+ALU_op2+sub;
// Falls B subtrahiert wird, muss 1==sub hinzuaddiert
    werden
// da im 2erkomplement gerechnet wird
always @(posedge clk)
    if (ue[2]) begin
    MAR = ALUout;
    C = move?B:ALUout;
    MDRw = B;
    if (aluop) ZF = (ALUout==0);
    end
end
```

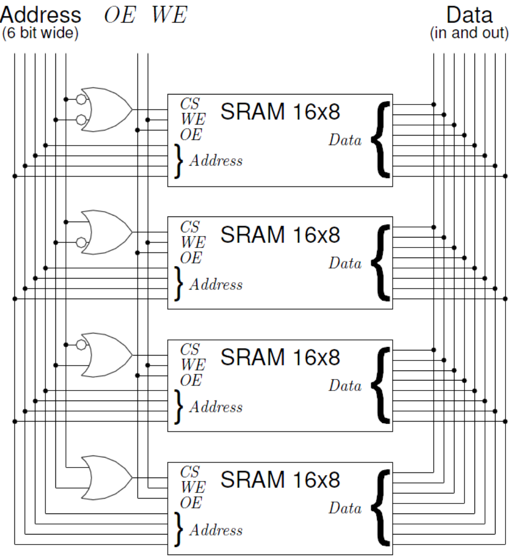


6 Code des Y86-Prozessors

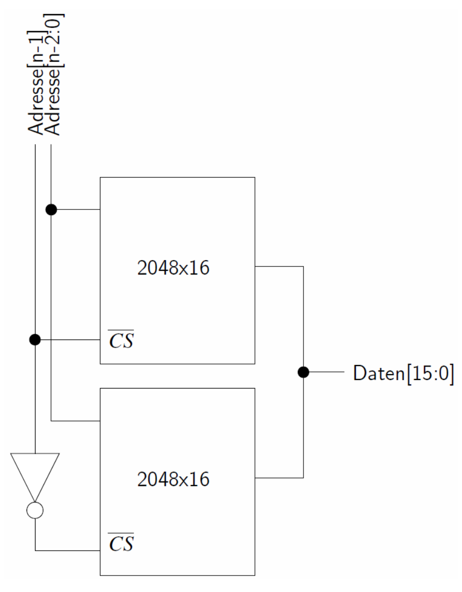
```
`timescale 1ns/1ps

module y86_seq(
  input clk,
  input rst,
  output [31:0] bus_A,
  input [31:0] bus_in,
  output [31:0] bus_out,
  output bus_WE, bus_RE,
  output [7:0] current_opcode);


  // global control

  reg [5:1] full;
  wire [4:0] ue={ full[4:1], full[5] };

  always @(posedge clk or posedge rst) begin
    if(rst)
       full='b10000;
     else
      full={ ue[4], ue[3], ue[2], ue[1], ue[0] };
  end


  // stage 1 IF

  reg [31:0] IR;

  always @(posedge clk)
    if(ue[0]) IR=bus_in;


  // stage 2 ID

  reg [31:0] IP, A, B;
  wire [31:0] Aop, Bop;
  wire [7:0] opcode=IR[7:0];
  wire [1:0] mod=IR[15:14];
  reg ZF;
  reg SF;
  reg OF;
  reg CF;
  wire load=opcode=='h8b && mod==1;
  wire move=opcode=='h89 && mod==3;
  wire store=opcode=='h89 && mod==1;
  wire memory=load || store;
  wire add=opcode=='h01;
  wire sub=opcode=='h29;
  wire halt=opcode=='hf4;
  wire aluop=add || sub;
  wire jnez=opcode=='h75;
  wire jge=opcode=='h7d;

  wire [4:0] RD=IR[10:8];
  wire [4:0] RS=IR[13:11];
  wire [4:0] Aad=memory?6:RD,
             Bad=RS;
  wire [31:0] distance={ { 24 { IR[15] } }, IR[15:8] };
  wire [31:0] displacement={{ 24 {IR[23]} }, IR[23:16] };
  wire btaken=(jnez && !ZF) || (jge && !(SF^OF));
  wire [1:0] length=memory                    ?3:
             (aluop || move || jnez || jge)?2:
                                            1;

  always @(posedge clk or posedge rst)
    if(rst)
        IP=0;
    else if(ue[1]) begin
      A=Aop;
      B=Bop;
      if(!halt) IP=IP+length;
      if(btaken) IP=IP+distance;
    end

  // stage 3 EX

  reg [31:0] MAR, MDRw, C;

  wire [31:0] ALU_op2=memory?displacement:sub?~B:B;
  wire [32:0] ALUout_carry= {1'b0, A} + {1'b0, ALU_op2}
        + {1'b0, sub};
  wire [31:0] ALUout = ALUout_carry[31:0];

  always @(posedge clk or posedge rst)
    if(rst) begin
       ZF=0;
        SF=0;
        OF=0;
        CF=0;
     end
    else if(ue[2]) begin
      MAR=ALUout;
      C=move?B:ALUout;
      MDRw=B;
      if(aluop) begin
          ZF=(ALUout==0);
          SF= ALUout[31];
          CF= ALUout_carry[32];
          OF= CF^SF^A[31]^ALU_op2[31];
        end
    end


  // stage 4 MEM

  reg [31:0] MDRr;

  always @(posedge clk)
    if(ue[3] && load) MDRr=bus_in;

  assign bus_A=ue[3]?MAR:ue[0]?IP:0;
  assign bus_RE=ue[0] || (ue[3] && load);


  // stage 5 WB

  reg [31:0] R[7:0];

  assign Aop=R[Aad];
  assign Bop=R[Bad];

  assign bus_WE=ue[3] && store;
  assign bus_out=MDRw;

  always @(posedge clk or posedge rst)
    if(rst) begin
       R[00]=0; R[01]=0; R[02]=0; R[03]=0;
       R[04]=0; R[05]=0; R[06]=0; R[07]=0;
     end
     else if(ue[4])
           if(aluop || move || load)
             R[load?RS:RD]=load?MDRr:C;

  assign current_opcode = opcode;

  // ... and now for something completely different.
  // synthesis translate_off

  wire [31:0] eax = R[0];
  wire [31:0] ecx = R[1];
  wire [31:0] edx = R[2];
  wire [31:0] ebx = R[3];
  wire [31:0] esp = R[4];
  wire [31:0] ebp = R[5];
  wire [31:0] esi = R[6];
  wire [31:0] edi = R[7];
  wire [7:0] regs = IR[15:8];

  // synthesis translate_on

endmodule
```

7 Memory

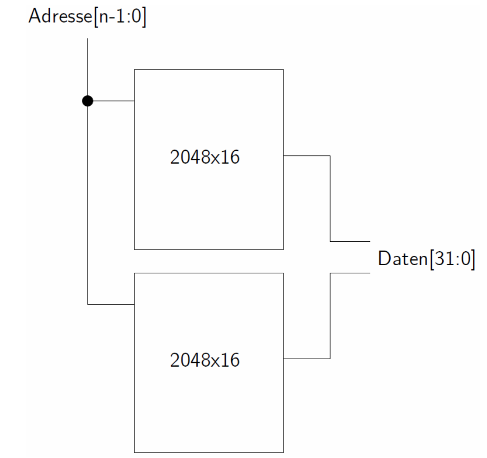
7.1 Kaskadierung von RAM-Chips



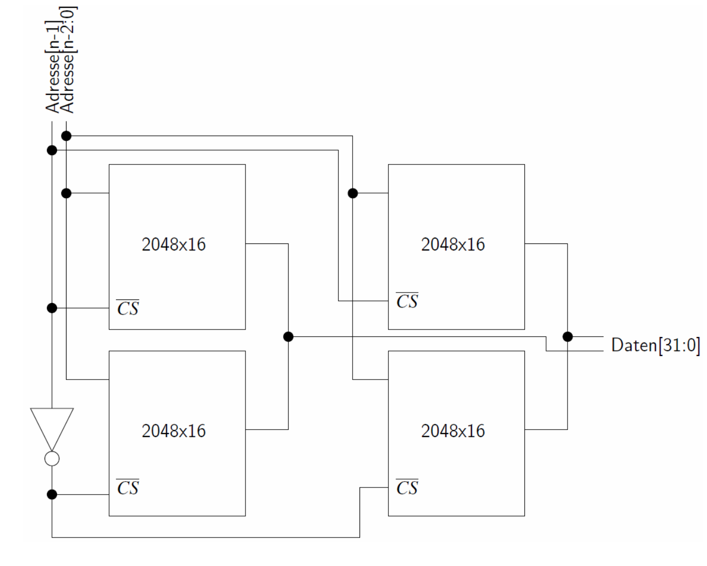
7.2 64 KBit (4096x16)



7.3 64 KBit (2048x32)



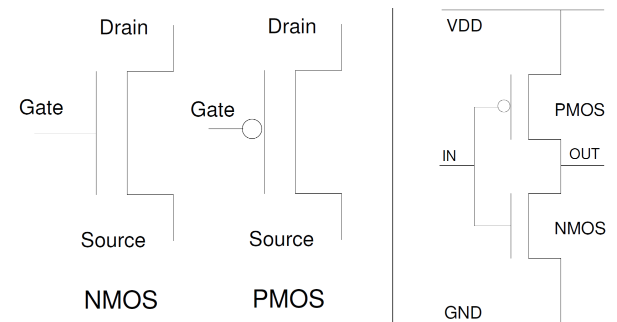
7.4 128 KBit (4096x32)



8 CMOS

Es gibt zwei Schalttypen, NMOS und PMOS, beide werden aus Halbleitertransistoren aufgebaut. Dabei kommt dotiertes Silizium zum Einsatz, entweder p-dotiert (mit einem Überschuss an positiver Ladung) oder n-dotiert (zu viel negative Ladung).

PMOS und NMOS Schalter dürfen nicht beliebig verbaut werden, ein NMOS-Transistor funktioniert nur zwischen `out` und `GND`, während ein PMOS-Transistor nur zwischen `VDD` und `out` eingesetzt werden darf.



9 Binärzahlen (2er-Komplement)

00000000	0	0x00	01000000	64	0x40	11111111	-1	255	0xff	10111111	-65	191	0xbf
00000001	1	0x01	01000001	65	0x41	11111110	-2	254	0xfe	10111110	-66	190	0xbe
00000010	2	0x02	01000010	66	0x42	11111101	-3	253	0xfd	10111101	-67	189	0xbd
00000011	3	0x03	01000011	67	0x43	11111100	-4	252	0xfc	10111100	-68	188	0xbc
00000100	4	0x04	01000100	68	0x44	11111011	-5	251	0xfb	10111011	-69	187	0xbb
00000101	5	0x05	01000101	69	0x45	11111010	-6	250	0xfa	10111010	-70	186	0xba
00000110	6	0x06	01000110	70	0x46	11111001	-7	249	0xf9	10111001	-71	185	0xb9
00000111	7	0x07	01000111	71	0x47	11111000	-8	248	0xf8	10111000	-72	184	0xb8
00001000	8	0x08	01001000	72	0x48	11110111	-9	247	0xf7	10110111	-73	183	0xb7
00001001	9	0x09	01001001	73	0x49	11110110	-10	246	0xf6	10110110	-74	182	0xb6
00001010	10	0x0a	01001010	74	0x4a	11110101	-11	245	0xf5	10110101	-75	181	0xb5
00001011	11	0x0b	01001011	75	0x4b	11110100	-12	244	0xf4	10110100	-76	180	0xb4
00001100	12	0x0c	01001100	76	0x4c	11110011	-13	243	0xf3	10110011	-77	179	0xb3
00001101	13	0x0d	01001101	77	0x4d	11110010	-14	242	0xf2	10110010	-78	178	0xb2
00001110	14	0x0e	01001110	78	0x4e	11110001	-15	241	0xf1	10110001	-79	177	0xb1
00001111	15	0x0f	01001111	79	0x4f	11110000	-16	240	0xf0	10110000	-80	176	0xb0
00010000	16	0x10	01010000	80	0x50	11101111	-17	239	0xef	10101111	-81	175	0xaf
00010001	17	0x11	01010001	81	0x51	11101110	-18	238	0xee	10101110	-82	174	0xae
00010010	18	0x12	01010010	82	0x52	11101101	-19	237	0xed	10101101	-83	173	0xad
00010011	19	0x13	01010011	83	0x53	11101100	-20	236	0xec	10101100	-84	172	0xac
00010100	20	0x14	01010100	84	0x54	11101011	-21	235	0xeb	10101011	-85	171	0xab
00010101	21	0x15	01010101	85	0x55	11101010	-22	234	0xea	10101010	-86	170	0xaa
00010110	22	0x16	01010110	86	0x56	11101001	-23	233	0xe9	10101001	-87	169	0xa9
00010111	23	0x17	01010111	87	0x57	11101000	-24	232	0xe8	10101000	-88	168	0xa8
00011000	24	0x18	01011000	88	0x58	11100111	-25	231	0xe7	10100111	-89	167	0xa7
00011001	25	0x19	01011001	89	0x59	11100110	-26	230	0xe6	10100110	-90	166	0xa6
00011010	26	0x1a	01011010	90	0x5a	11100101	-27	229	0xe5	10100101	-91	165	0xa5
00011011	27	0x1b	01011011	91	0x5b	11100100	-28	228	0xe4	10100100	-92	164	0xa4
00011100	28	0x1c	01011100	92	0x5c	11100011	-29	227	0xe3	10100011	-93	163	0xa3
00011101	29	0x1d	01011101	93	0x5d	11100010	-30	226	0xe2	10100010	-94	162	0xa2
00011110	30	0x1e	01011110	94	0x5e	11100001	-31	225	0xe1	10100001	-95	161	0xa1
00011111	31	0x1f	01011111	95	0x5f	11100000	-32	224	0xe0	10100000	-96	160	0xa0
00100000	32	0x20	01100000	96	0x60	11011111	-33	223	0xdf	10011111	-97	159	0x9f
00100001	33	0x21	01100001	97	0x61	11011110	-34	222	0xde	10011110	-98	158	0x9e
00100010	34	0x22	01100010	98	0x62	11011101	-35	221	0xdd	10011101	-99	157	0x9d
00100011	35	0x23	01100011	99	0x63	11011100	-36	220	0xdc	10011100	-100	156	0x9c
00100100	36	0x24	01100100	100	0x64	11011011	-37	219	0xdb	10011011	-101	155	0x9b
00100101	37	0x25	01100101	101	0x65	11011010	-38	218	0xda	10011010	-102	154	0x9a
00100110	38	0x26	01100110	102	0x66	11011001	-39	217	0xd9	10011001	-103	153	0x99
00100111	39	0x27	01100111	103	0x67	11011000	-40	216	0xd8	10011000	-104	152	0x98
00101000	40	0x28	01101000	104	0x68	11010111	-41	215	0xd7	10010111	-105	151	0x97
00101001	41	0x29	01101001	105	0x69	11010110	-42	214	0xd6	10010110	-106	150	0x96
00101010	42	0x2a	01101010	106	0x6a	11010101	-43	213	0xd5	10010101	-107	149	0x95
00101011	43	0x2b	01101011	107	0x6b	11010100	-44	212	0xd4	10010100	-108	148	0x94
00101100	44	0x2c	01101100	108	0x6c	11010011	-45	211	0xd3	10010011	-109	147	0x93
00101101	45	0x2d	01101101	109	0x6d	11010010	-46	210	0xd2	10010010	-110	146	0x92
00101110	46	0x2e	01101110	110	0x6e	11010001	-47	209	0xd1	10010001	-111	145	0x91
00101111	47	0x2f	01101111	111	0x6f	11010000	-48	208	0xd0	10010000	-112	144	0x90
00110000	48	0x30	01110000	112	0x70	11001111	-49	207	0xcf	10001111	-113	143	0x8f
00110001	49	0x31	01110001	113	0x71	11001110	-50	206	0xce	10001110	-114	142	0x8e
00110010	50	0x32	01110010	114	0x72	11001101	-51	205	0xcd	10001101	-115	141	0x8d
00110011	51	0x33	01110011	115	0x73	11001100	-52	204	0xcc	10001100	-116	140	0x8c
00110100	52	0x34	01110100	116	0x74	11001011	-53	203	0xcb	10001011	-117	139	0x8b
00110101	53	0x35	01110101	117	0x75	11001010	-54	202	0xca	10001010	-118	138	0x8a
00110110	54	0x36	01110110	118	0x76	11001001	-55	201	0xc9	10001001	-119	137	0x89
00110111	55	0x37	01110111	119	0x77	11001000	-56	200	0xc8	10001000	-120	136	0x88
00111000	56	0x38	01111000	120	0x78	11000111	-57	199	0xc7	10000111	-121	135	0x87
00111001	57	0x39	01111001	121	0x79	11000110	-58	198	0xc6	10000110	-122	134	0x86
00111010	58	0x3a	01111010	122	0x7a	11000101	-59	197	0xc5	10000101	-123	133	0x85
00111011	59	0x3b	01111011	123	0x7b	11000100	-60	196	0xc4	10000100	-124	132	0x84
00111100	60	0x3c	01111100	124	0x7c	11000011	-61	195	0xc3	10000011	-125	131	0x83
00111101	61	0x3d	01111101	125	0x7d	11000010	-62	194	0xc2	10000010	-126	130	0x82
00111110	62	0x3e	01111110	126	0x7e	11000001	-63	193	0xc1	10000001	-127	129	0x81
00111111	63	0x3f	01111111	127	0x7f	11000000	-64	192	0xc0	10000000	-128	128	0x80

10 Arithmetik

10.1 1er-Komplement

$$\langle s_n d_{n-1} \dots d_0 \rangle_{1er} := \begin{cases} \sum_{i=0}^{n-1} d_i \cdot 2^i & s_n = 0 \\ -\sum_{i=0}^{n-1} (1-d_i) \cdot 2^i & s_n = 1 \end{cases}$$

10.2 2er-Komplement

$$[\![s_n d_{n-1} \dots d_0]\!] = \begin{cases} \sum_{i=0}^{n-1} d_i \cdot 2^i & s_n = 0 \\ -(1 + \sum_{i=0}^{n-1} (1-d_i) \cdot 2^i) & s_n = 1 \end{cases}$$

$$= s_n \cdot -2^n + \sum_{i=0}^{n-1} d_i \cdot 2^i$$

10.3 Lemma

$$2^n = 1 + \sum_{i=0}^{n-1} 1 \cdot 2^i$$

10.4 Beweis 2er-Komplement-Darstellungen

$$\begin{aligned} [\![1\ d_{n-1} \dots d_0]\!] &= 1 \cdot -2^n + \sum_{i=0}^{n-1} d_i \cdot 2^i \\ &= -\left(2^n - \sum_{i=0}^{n-1} d_i \cdot 2^i\right) \\ &= -\left(2^n + \sum_{i=0}^{n-1} -d_i \cdot 2^i\right) \\ &= -\left(1 + \sum_{i=0}^{n-1} 1 \cdot 2^i + \sum_{i=0}^{n-1} -d_i \cdot 2^i\right) \\ &= -\left(1 + \sum_{i=0}^{n-1} (1 \cdot 2^i - d_i \cdot 2^i)\right) \\ &= -\left(1 + \sum_{i=0}^{n-1} (1 - d_i) \cdot 2^i\right) \end{aligned}$$

10.5 Halbaddierer

$$\begin{aligned} s &\equiv (a+b) \bmod 2 &\equiv a \oplus b \\ o &\equiv (a+b) \operatorname{div} 2 &\equiv a \wedge b \end{aligned}$$

10.6 Volladdierer

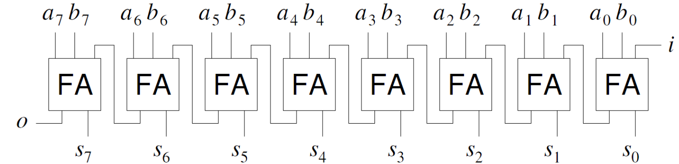
$$\begin{aligned} s &\equiv (a+b+i) \bmod 2 &\equiv a \oplus b \oplus i \\ o &\equiv (a+b+i) \operatorname{div} 2 &\equiv a \cdot b + a \cdot i + b \cdot i \end{aligned}$$

10.7 Überlauf

$$[\![a]\!] + [\![b]\!] \notin \{-2^{n-1}, \dots, 2^{n-1} - 1\} \iff c_{n-1} \oplus c_n$$

10.8 Ripple-Carry Addierer

Einfacher Addierer mit $O(n)$ Zeit und Platz.



10.9 Booth Recoding - Multiplikation zweier Binärzahlen $(A \cdot B)$

i. Wähle kürzere der beiden Zahlen als A, wähle P gleich lang wie B

ii. Erstelle Tabelle $P_{m,\dots,0} \mid A_{n,\dots,0} \mid A_{-1} = 0$

iii. Wende folgende *Regeln* an, starte bei $i = 0$, Tue bei jedem Schritt: $i := i + 1$:

A_i	A_{i-1}	ToDo
0	0	-
0	1	addiere B zu P
1	0	addiere -B zu P
1	1	-

iv. Shifte arithmetisch nach rechts (d.h. Vorzeichen nachschieben)

v. Wiederhole (3) und (4) n (Länge A) mal