

5 Übersicht

6 Prozessorarchitektur – Überblick

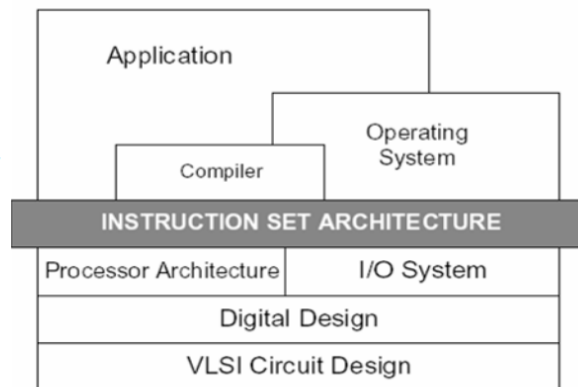
6.1 Von-Neumann-Architektur / Harvard-Architektur

6.2 Befehlsabarbeitung

6.3 Instruktionssatzarchitektur

Bedeutung des Befehlssatzes (ISA)

- Befehlssatz = Instruction Set Architektur (ISA)
- Der ISA Level ist das Interface zwischen der Software und der Hardware
- Der ISA Level definiert die Sprache, die sowohl von der Software als auch von der Hardware verstanden werden muß.

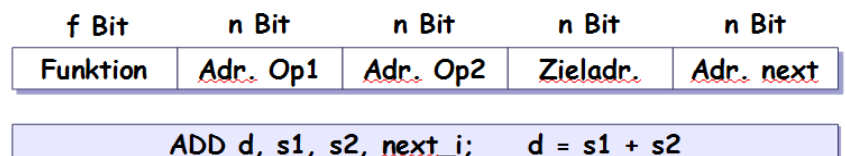


6.3.1 Einteilung der Befehlssätze – Anzahl der Adressen

4-Adress-Befehlsformat

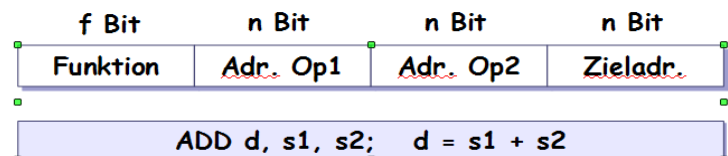
allgemeinste Form

- schwierig zu programmieren
- wird für Mikrocode verwendet



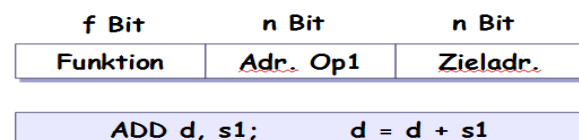
3-Adress-Befehlsformat

- Standard bei RISC
- Adressen brauchen Platz (32 Bit)



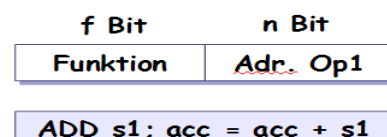
2-Adress-Befehlsformat

- Standard bei 8, 16 BIT uPs
- Intel IA32
- komprimierte 16 Bit Befehlssätze



1-Adress-Befehlsformat

- Zielregister implizit (Akkumulator)



0-Adress-Befehlsformat

- Alle Register implizit (Stack)
- zB: Java VM
- zusätzliche Befehle zum Laden und Speichern des Stacks notwendig

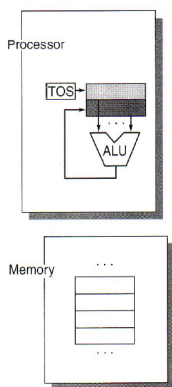
f Bit

Funktion

ADD ;
~~top_of_stack = top_of_stack + next_on_stack~~

6.3.2 Einteilung der Befehlssätze – Operanden

Stack Architektur:



Die Code-Sequenz

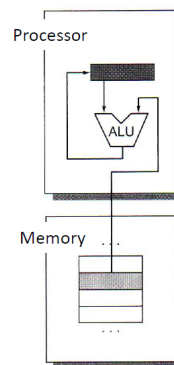
 $C = A + B$

wird wie folgt bearbeitet

Push A
 Push B
 Add
 Pop C

Bildquelle: John L. Hennessy und David A. Patterson, „Computer Architecture“, Fifth Edition, 2012
 Grundlagen der Rechnerarchitektur - Assembler

Akkumulator Architektur:



Die Code-Sequenz

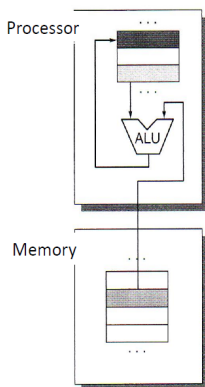
 $C = A + B$

wird wie folgt bearbeitet

Load A
 Add B
 Store C

Bildquelle: John L. Hennessy und David A. Patterson, „Computer Architecture“, Fifth Edition, 2012
 Grundlagen der Rechnerarchitektur - Assembler

Register-Memory Architektur:



Die Code-Sequenz

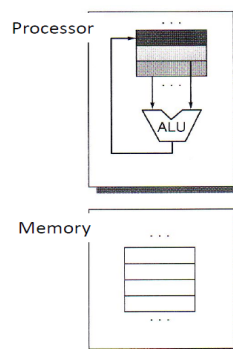
 $C = A + B$

wird wie folgt bearbeitet

Load R1, A
 Add R3, R1, B
 Store R3, C

Bildquelle: John L. Hennessy und David A. Patterson, „Computer Architecture“, Fifth Edition, 2012
 Grundlagen der Rechnerarchitektur - Assembler

Register - Register Architektur:



Die Code-Sequenz

 $C = A + B$

wird wie folgt bearbeitet

Load R1, A
 Load R2, B
 Add R3, R1, R2
 Store R3, C

Bildquelle: John L. Hennessy und David A. Patterson, „Computer Architecture“, Fifth Edition, 2012
 Grundlagen der Rechnerarchitektur - Assembler

6.3.3 Beschreibung eines Befehls: RTL

6.3.3.1 Register Transfer

Ein digitales System besteht aus kombinatorischen und sequentiellen Logikblöcken; beschrieben kann es werden durch die Register und durch die Operationen, die mit den Daten der Register ausgeführt werden = Mikrooperation.

Wenn der Fokus liegt auf:

- Systemregister
- Datentransformationen in ihnen
- Datentransfer zwischen den Registern

verwendet man oft die RTL-Notation (Register Transfer Language).

Es gibt viele RTLs. Sie sind symbolische Sprachen und ein gebräuchliches Mittel, um die interne Organisation von Computern zu beschreiben und sie erleichtern den Designprozess.

Register: A, IP, PC wenn ganzes Register gemeint, sonst A(7:1) für Teile, A(0) für Einzelbit

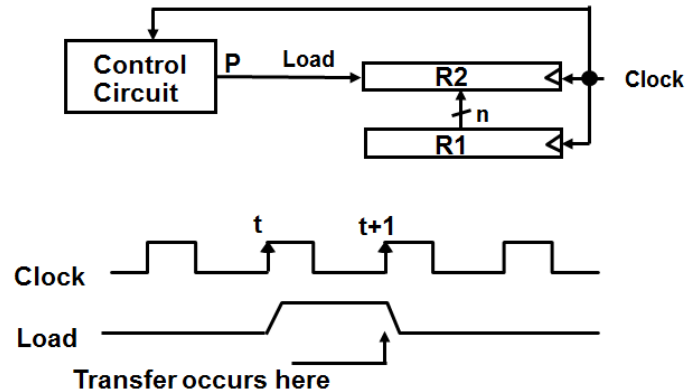
Gesteuerter Transfer:

P: $R2 \leftarrow R1$ (Inhalt von R1 wird in R2 gespeichert, falls P=1)

Mehrere Transfers mit derselben Taktflanke werden in eine Zeile geschrieben:

P1: $MAR \leftarrow A$, P2: $A \leftarrow B$

$A \leftarrow 12$ (Konstante laden)



6.3.3.2 Bus Transfer

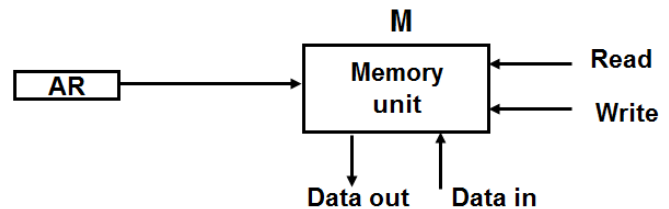
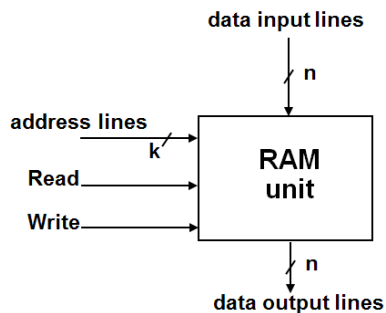
Daten werden normalerweise über Busse transportiert zB:

P: $DBUS \leftarrow R1$, $R2 \leftarrow DBUS$ (Datentransfer über Bus wird ausdrücklich angeschrieben),

oder kurz: P: $R2 \leftarrow R1$

6.3.3.3 Memory Transfer

Speicherdarstellung: Der Speicher wird normalerweise nicht direkt über den Adressbus angesteuert sondern über ein Adressregister:



Leseoperation: $R1 \leftarrow M[AR]$; Schreiboperation: $M[AR] \leftarrow R2$

6.3.3.4 Arithmetische, logische und shift Mikrooperationen

Typische Arithmetik-Mikrooperationen

$R3 \leftarrow R1 + R2$; $R3 \leftarrow R1 + R2'' + 1$ (Subtraktion); $R1 \leftarrow R1 + 1$ (Inkrement)

Typische Logikoperationen:

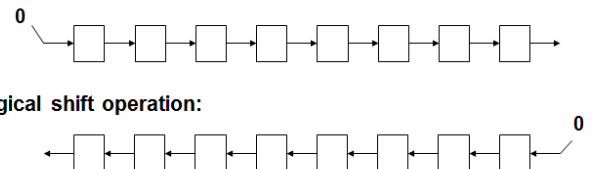
x	y	Boolean Function	Micro-Operations	Name
0	0	$F0 = 0$	$F \leftarrow 0$	Clear
0	0	$F1 = xy$	$F \leftarrow A \wedge B$	AND
0	0	$F2 = xy'$	$F \leftarrow A \wedge B'$	
0	0	$F3 = x$	$F \leftarrow A$	Transfer A
0	1	$F4 = x'y$	$F \leftarrow A' \wedge B$	
0	1	$F5 = y$	$F \leftarrow B$	Transfer B
0	1	$F6 = x \oplus y$	$F \leftarrow A \oplus B$	Exclusive-OR
0	1	$F7 = x + y$	$F \leftarrow A \vee B$	OR
1	0	$F8 = (x + y)'$	$F \leftarrow (A \vee B)'$	NOR
1	0	$F9 = (x \oplus y)'$	$F \leftarrow (A \oplus B)'$	Exclusive-NOR
1	0	$F10 = y'$	$F \leftarrow B'$	Complement B
1	0	$F11 = x + y'$	$F \leftarrow A \vee B$	
1	1	$F12 = x'$	$F \leftarrow A'$	Complement A
1	1	$F13 = x' + y$	$F \leftarrow A' \vee B$	
1	1	$F14 = (xy)'$	$F \leftarrow (A \wedge B)'$	NAND
1	1	$F15 = 1$	$F \leftarrow \text{all 1's}$	Set to all 1's

Logikoperationen zur Bitmanipulation:

– Selective-set	$A \leftarrow A + B$
– Selective-complement	$A \leftarrow A \oplus B$
– Selective-clear	$A \leftarrow A \cdot B'$
– Mask (Delete)	$A \leftarrow A \cdot B$
– Clear	$A \leftarrow A \oplus B$
– Insert	$A \leftarrow (A \cdot B) + C$
– Compare	$A \leftarrow A \oplus B$

Shift und Rotate Operationen:

- Logische shift Operation füllt freies Bit mit 0 an (shl, shr);
- arithmetisches shift left ashl zieht 0 nach (msb und msb-1 müssen gleich sein);
- ashr verdoppelt Vorzeichenbit.
- Rotate oder circular shift (cil, cir);
Achtung: Behandlung des carry!
- In a logical shift the serial input to the shift is a 0.
- A right logical shift operation:
- A left logical shift operation:
- In a Register Transfer Language, the following notation is used
 - *shl* for a logical shift left
 - *shr* for a logical shift right
 - Examples:
 - » $R2 \leftarrow shr R2$
 - » $R3 \leftarrow shl R3$



6.3.4 Ausführung eines Befehls

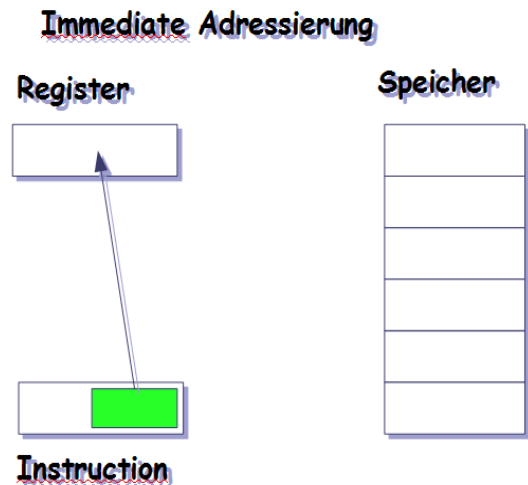
Befehlszyklus

Beispiele bei vNeuman Architektur und CISC Maschine

6.3.5 Adressierung der Operanden – Adressierungsarten

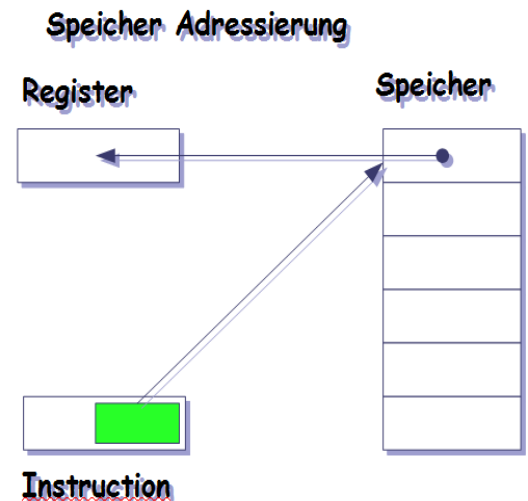
Unmittelbare Adressierung (Immediate)

- Der Adressteil der Instruktion enthält den Operanden selbst, anstatt eines Verweises.
- Solche Operanden werden als Direktoperanden (Immediate) bezeichnet
- ARM:
 - `mov R0, #8` ; Lade in das Register R0 den Wert 8
- Bei Intel Prozessoren kann der Immediate Wert eine Größe von 32 Bit besitzen
- Bei den meisten Risc Prozessoren ist der Immediate Wert deutlich kleiner (12 Bit bei ARM)
- Da der Wert im Befehl enthalten ist, ist für immediate Adressierung kein weiterer Speicherzugriff erforderlich



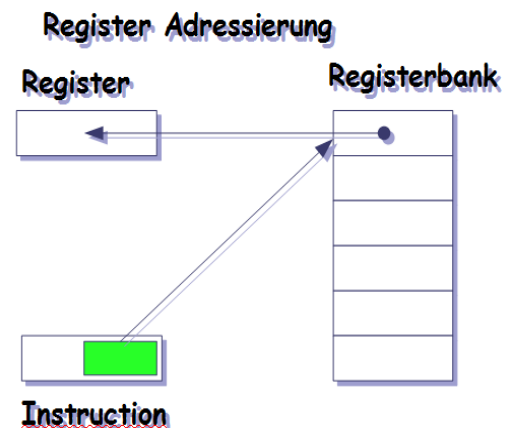
Direktadressierung (Direct Addressing)

- Die Adresse von der ein Wert geladen werden soll, befindet sich im Opcode.
- Deshalb nur für globale Variablen anwendbar, da Instruktion immer auf gleiche Speicherzeile zugreift
- Nachteil: die Adresse kann zur Laufzeit nicht mehr geändert werden
- Risc Prozessoren können keine 32 Bit Adressen in ihrem Opcode unterbringen, daher beherrschen sie diese Adressierungsart nicht
- Beispiel Intel
 - `mov ax, hugo`
 - ...
 - `hugo dw ?`



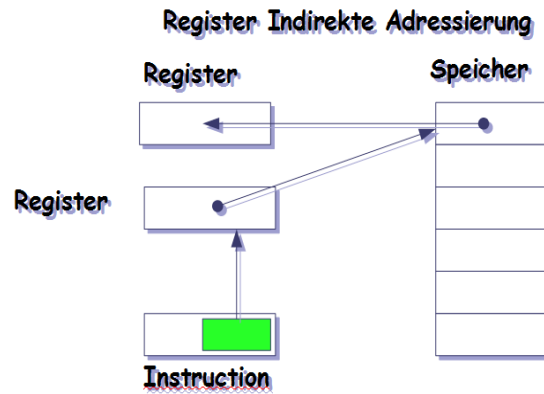
Registeradressierung

- Register Adressierung ist vom Konzept identisch zur direkten Adressierung, nur daß jetzt statt der Speicheradresse eine Registeradresse benutzt wird.
- In Registern sollten die am häufigsten verwendeten Variablen abgelegt werden, da Register vielfach schneller als Hauptspeicher sind
- Load/Store-Architekturen nutzen fast nur diesen Registermode (außer es muss vom oder zum Speicher transferiert werden)
- Beispiele:
 - Intel: `add ax, bx`
 - ARM: `add r0, r1, r2`
- In Load Store Architekturen nutzen alle Befehle diese Adressierungsart, außer den Load und Store Befehlen selbst. Aber auch bei diesen Befehlen ist das Ziel oder die Quelle ein Register



Register indirekte Adressierung

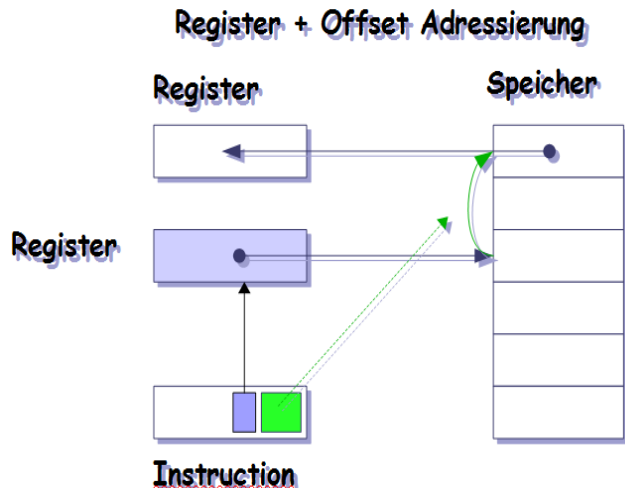
- Die Adresse wird nicht direkt angegeben, sondern indirekt über ein Register
- Das Register enthält somit einen Pointer auf eine Speicherzelle
- Der Vorteil dieser Adressierungsart ist, daß man den gesamten Speicherbereich adressieren kann, ohne daß man eine Adresse in der Instruktion unterbringen muß.
- Beispiele:
 - Intel: `mov ax, [bx]`
 - ARM: `ldr r0, [r1]`
- Diese Adressierungsart ist in Load und Store Architekturen nur in den Load und Store Befehlen erlaubt



Benutzt einen Registerwert (**Basisregister**) als Speicheradresse zum Laden oder Speichern des Wertes an dieser Adresse

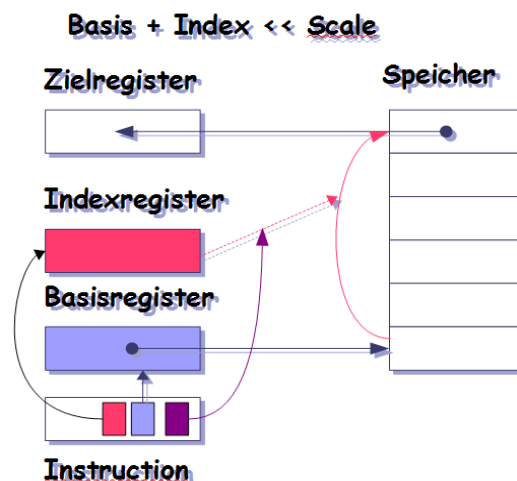
Indizierte Adressierung

- Der Speicher wird durch Angabe eines Registers und eines konstanten Offsets adressiert
- Häufig benutzt bei Arrayzugriffen, wie $A = B[i]$ (`MOV R1, B[R2]`)
- In dieser Adressierungsart kommt auch einer der Operanden aus dem Speicher, aber die Adresse des Speichers steht in einem Register. (Adressierung über Pointer)



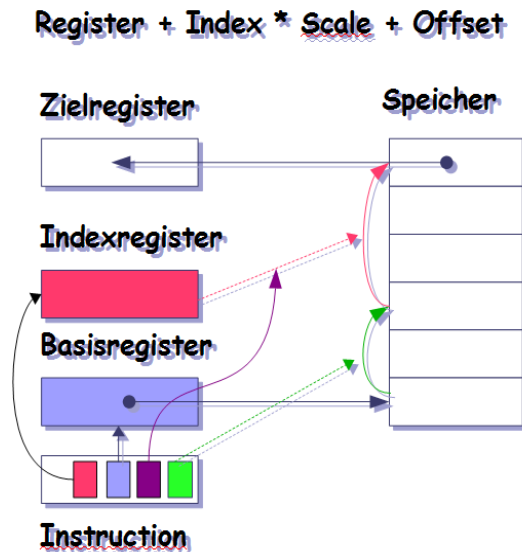
Basisindizierte Adressierung

- Moderne Prozessoren besitzen häufig noch eine Adressierungsart in der zu einem Basisregister noch ein Indexregister addiert wird.
- Eines der Register stellt die Basis dar und ein anderes den Index
- Anwendungen
 - Vektoradressierung, Stackadressierung
- ARM: `ldr r1, [r2, r0, LSL #2]`



Dies ist die komplizierteste Adressierung die ein ARM Prozessor beherrscht. Dabei wird das **Basisregister** auf den A-Bus gelegt und das **Indexregister** wird auf dem B-Bus über den **Barrelshifter** zur ALU geführt. Diese berechnet daraus die endgültige Adresse

Basisindizierte Adressierung mit Offset



Diese Adressierungsform findet man bei den x86 Prozessoren als kompliziertesten Adressierungsmechanismus

Stapeladressierung

- Hier ist gar keine Adressangabe notwendig
- Somit sind die Instruktionen sehr kurz
- Die Stapeladressierung arbeitet mit der umgekehrten polnischen Notation (Postfix)

7 Mikroarchitektur

7.1 Elemente einer RA

7.1.1 Bussystem

- Point to Point mit Bündel von Leitungen
- Multiplexer an den Ausgängen zum Bus hin
- Tristate Logik an den Ausgängen

```
entity tribuf8 is
    Port ( a : in  STD_LOGIC_VECTOR (7 downto 0);
          oe : in  STD_LOGIC;
          b : out  STD_LOGIC_VECTOR (7 downto 0));
end tribuf8;
```

```
architecture Behavioral of tribuf8 is
begin
    b <= a when oe='1' else (others => 'Z');
end Behavioral;
```

7.1.2 Register

```
entity reg8 is
    Port ( clk : in  STD_LOGIC;
          We : in  STD_LOGIC;
          Oe : in  STD_LOGIC;
          Din : in  STD_LOGIC_VECTOR (7 downto 0);
```

```
Dout : out  STD_LOGIC_VECTOR (7 downto 0);
rst : in   STD_LOGIC);
end reg8;

architecture Behavioral of reg8 is

signal q1: STD_LOGIC_VECTOR (7 downto 0);

begin
p_a: process (clk, rst)
begin
    if rst = '1' then q1 <= (others=>'0');
    elsif (clk'event and clk = '1') then
        if We = '1' then q1 <= Din;
        end if;
    end if;
end process;

Dout<= q1 when Oe = '1' else (others=>'Z');

end Behavioral;
```

7.1.3 RAM

Ram ohne Adressregister:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity MEM256x8 is
    Port ( clk : in  STD_LOGIC;
          MEMw : in  STD_LOGIC;
          OE  : in  STD_LOGIC;
          adr : in  STD_LOGIC_VECTOR (7 downto 0);
          din : in  STD_LOGIC_VECTOR (7 downto 0);
          dout : out STD_LOGIC_VECTOR (7 downto 0));
end MEM256x8;

architecture Behavioral of MEM256x8 is

type RAM_TYPE is array (0 to 255) of std_logic_vector (7 downto 0);

signal D_RAM : RAM_TYPE := (others => x"00");

begin

ram_p: process (clk)
begin
    if clk'event and clk = '1' then
        if MEMw = '1' then
            D_RAM(to_integer(unsigned(adr))) <= din;
        end if;
    end if;
end process;

dout <= D_RAM(to_integer(unsigned(adr))) when OE='1' else (others=>'Z');

end Behavioral;
```


7.1.4 ROM

Asynchrones ROM:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity rom256x8 is
    Port ( adr : in  STD_LOGIC_VECTOR (7 downto 0);
          data : out STD_LOGIC_VECTOR (7 downto 0));
end rom256x8;

architecture Behavioral of rom256x8 is

    type ROM_TYPE is array (0 to 255) of std_logic_vector (7 downto 0);

    constant ROM : ROM_TYPE :=
        ( x"00", x"00", x"00", x"00", x"04", x"00", x"06", x"00", x"08", x"00", others =>
          x"00" );

    begin
        data <= std_logic_vector( ROM(to_integer( unsigned(adr))) );
    end Behavioral;
```

7.1.5 Rechenwerk

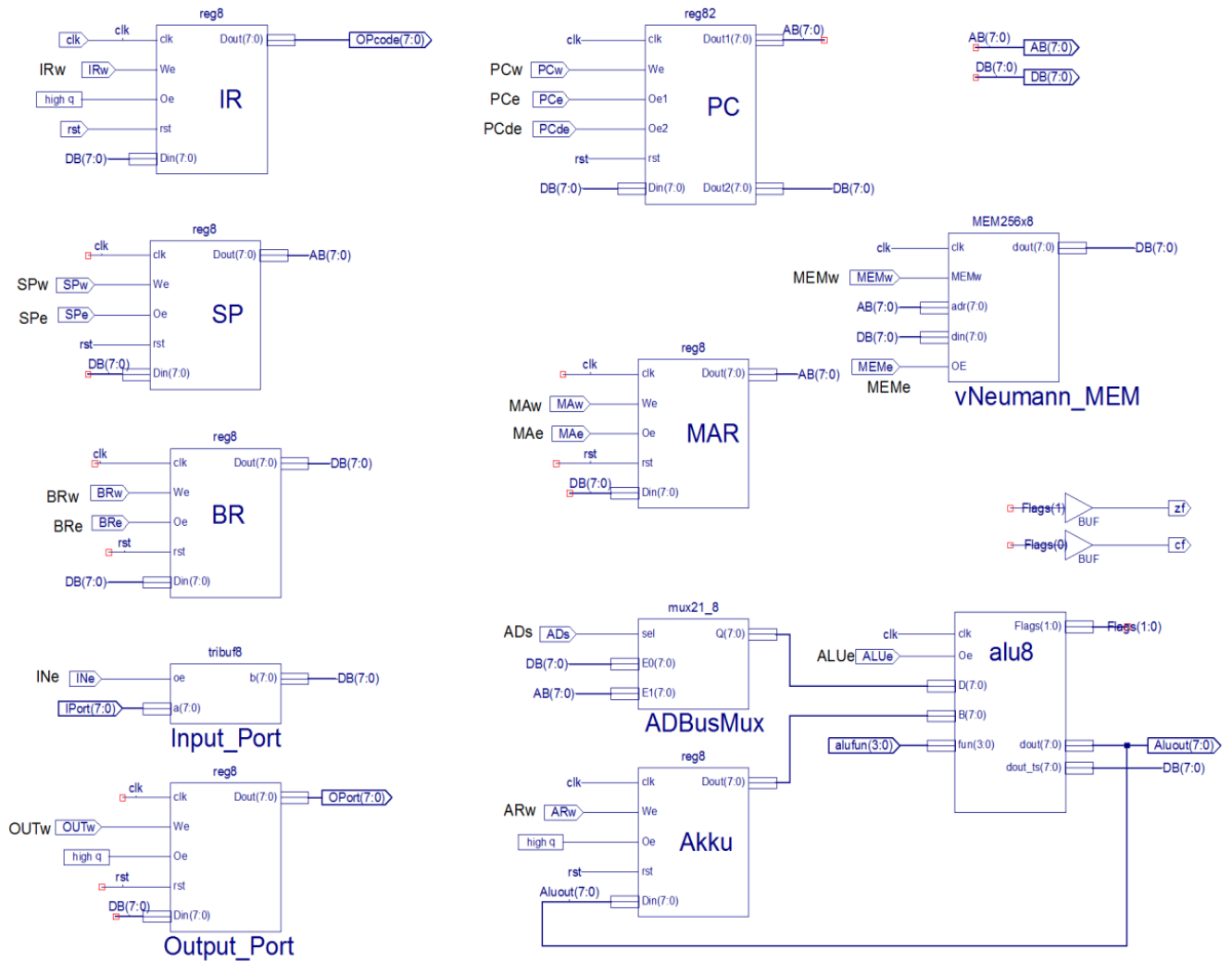
7.1.6 Steuerwerk

7.2 CISC - CPU

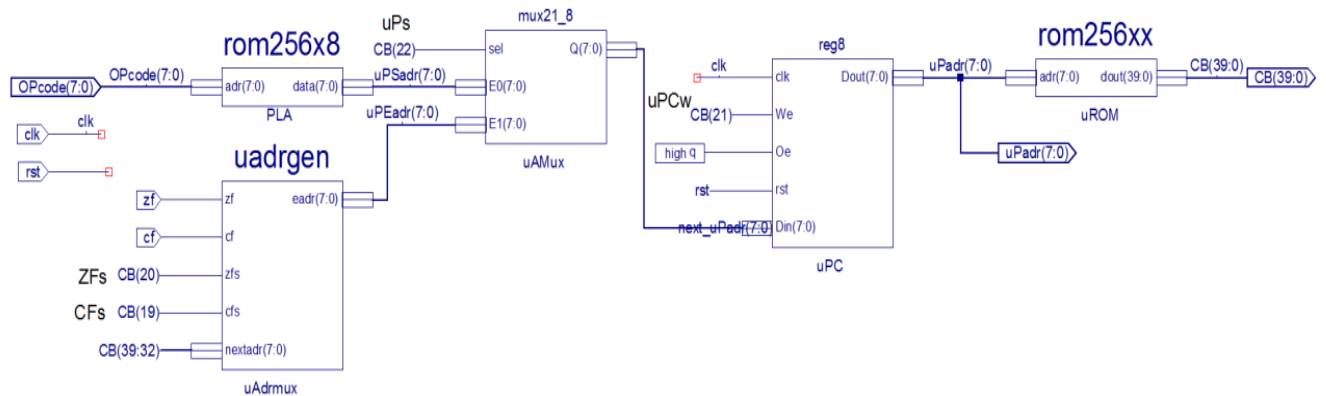
7.2.1 Entwicklung Datenpfad CISC-CPU

Instruction fetch: IRw: $IR \leftarrow MEM[DB]$, Pce: $DB \leftarrow PC$

Incr. PC: Pcw, ALUe, ALUfun: $PC \leftarrow DB + 1$, PCe: $DB \leftarrow PC$



7.2.2 Mikroprogrammiertes Leitwerk CISC-CPU



7.2.3 Befehlsdefinitionen (Mikroprogramm)

Beispiele siehe Mitschrift

8 Eingebettete Systeme (embedded systems)

Aus: Ralf Gessler: Entwicklung Eingebetteter Systeme (Vergleich von Entwicklungsprozessen für FPGA- und Mikroprozessor-Systeme Entwurf auf Systemebene) Springer ebook

„Wenn man von Computern spricht, denkt man zunächst an Geräte wie Personal Computer, Laptops, Workstations oder Großrechner. Aber es gibt auch noch andere, weiter verbreitete Rechnersysteme: die eingebetteten Systeme. Das sind Rechenmaschinen, die in elektrischen Geräten „eingebettet“ sind, z. B. in Systeme Kaffeemaschinen, CD-, DVD-Spielern oder Mobiltelefonen. Verglichen mit Millionen produzierter Desktop-Systeme werden Milliarden eingebetteter Systeme pro Jahr hergestellt. Vielfach findet man bis zu 50 Geräte pro Haushalt und Automobil.

Im Folgenden gilt diese Definition:

- Eingebettete Systeme: Rechenmaschinen, die für den Anwender weitgehend unsichtbar in einem elektrischen Gerät „eingebettet“ sind.

Eingebettete Systeme weisen folgende Merkmale auf:

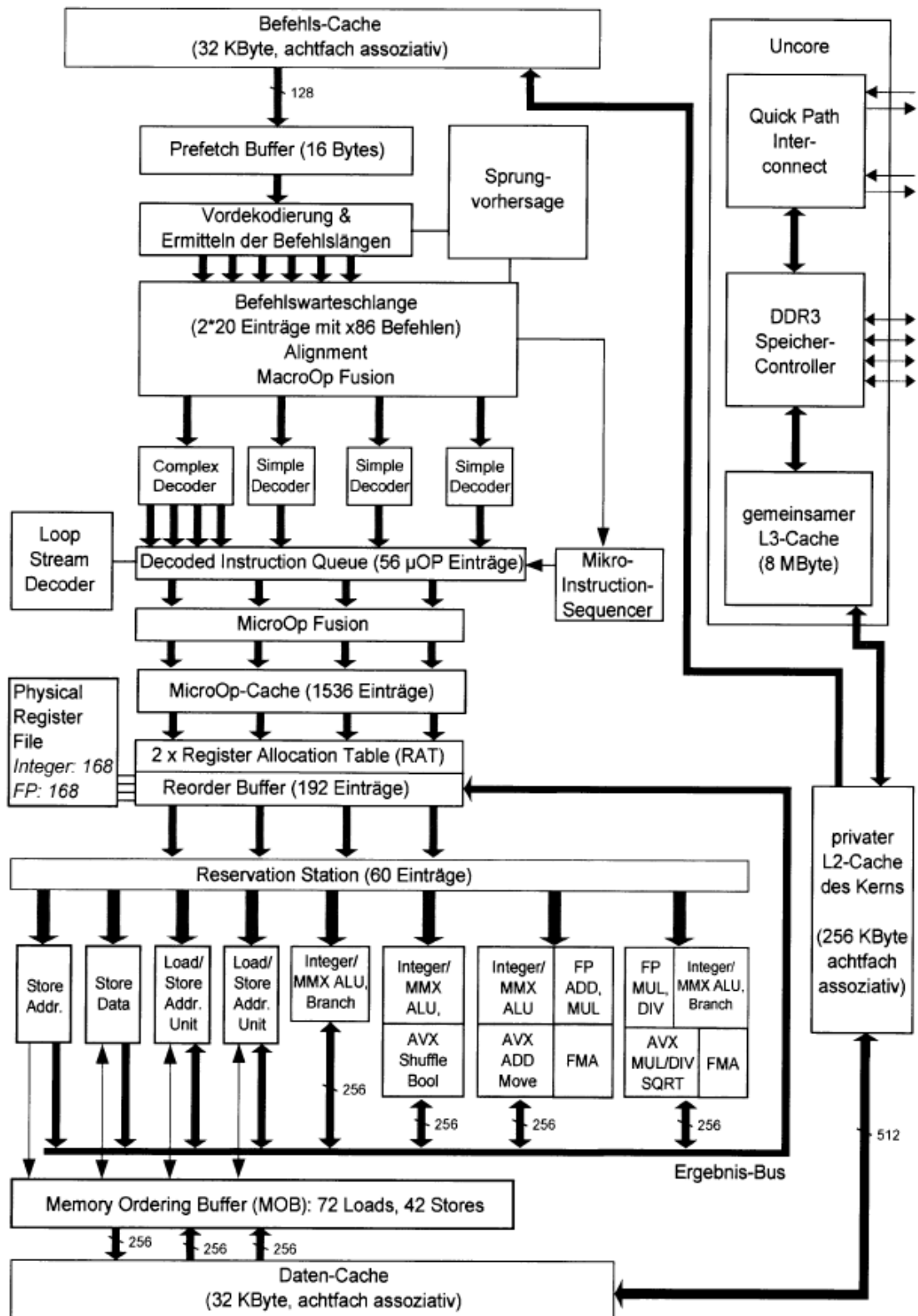
1. Ein eingebettetes System führt eine Funktion (wiederholt) aus.
2. Es gibt strenge Randbedingungen bezüglich Kosten, Energieverbrauch, Abmessungen usw.
3. Sie reagieren auf ihre Umwelt in Echtzeit

8.1 Prozessor Funktionsbeschreibung

tbd

8.2 Strukturbeschreibung

Mikroarchitektur eines Intel I7 (Haswell):



8.3 Entwurf eines Prozessors

8.3.1 Entwurfsziele

Das Ziel eines Entwurfs kann z. B. die Optimierung für eine spezielle Anwendung, an Stelle eines Universalprozessors, sein. Im vorliegenden Fall wird ein möglichst gut nachvollziehbares einfaches Design eines Prozessors angestrebt.

Dazu bietet sich das RISC-Konzept als Basis an. Für den Hardwareentwurf eines solchen Prozessors wird die Einhaltung von vier elementaren Grundprinzipien vorgeschlagen:

1. Einfachheit begünstigt Regelmäßigkeit,
2. kleiner ist schneller,
3. Optimierung auf den am häufigsten vorkommenden Fall,
4. ein guter Entwurf erfordert gute Kompromisse.

Das erste Prinzip wird z. B. durch die Verwendung von Befehlen im Drei-Adress-Format erfüllt. Damit lassen sich effizient in einer höheren Programmiersprache formulierte Gleichungen in die Maschinensprache des Prozessors überführen.

Prinzip 2 entspricht dem RISC-Paradigma, das sich der Entwickler einer Befehlssatzarchitektur auf das Wesentliche konzentrieren muss. Besser wenige Befehle, diese aber so effektiv wie möglich umgesetzt. Je komplexer der Gesamtaufbau ist, umso länger sind die Signalwege und desto mehr Multiplexer befinden sich im Datenpfad. Selbst bei der Anzahl der Arbeitsregister kommt das zweite Prinzip zum Einsatz. Je mehr Register vorhanden sind, umso aufwändiger wird auch deren Auswahl. Als Kompromiss hat sich bei einem 32 Bit breitem Befehlswort eine Größe von 32 Registern herausgestellt. Dadurch sind pro Register fünf Adressleitungen (ld(32): 5) nötig. Bei einem Drei-Adress-Format werden in diesem Fall bereits 15 Bits des Befehlswortes für die Auswahl der Register benötigt.

Das Prinzip 3 beschreibt den Anspruch, das genau die Operationen, die besonders oft benötigt werden, vom Prozessor als Maschinenbefehle angeboten werden. Dadurch wird die Anzahl der für ein Programm benötigten Befehle verringert und somit die Ausführungsgeschwindigkeit erhöht.

Prinzip 4 fordert dazu auf, Kompromisse beim Design einzugehen. Beispielsweise würde ein einheitliches Befehlsformat über den gesamten Befehlssatz, bei der jeder Befehl über alle möglichen Beschreibungsfelder verfügt, zu einer ineffizienten Ausnutzung der verfügbaren Bits im Befehlswort führen. Hier besteht der Kompromiss in der Einteilung in wenige Befehlstypen, welche in ihrem Aufbau für die jeweilige Befehlsgruppe optimiert sind.

8.3.2 Vergleich ARM – MIPS

Grundlegendes:

	ARM	MIPS
Date announced	1985	1985
Instruction size (bits)	32	32
Address space (size, model)	32 bits, flat	32 bits, flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Integer registers (number, model, size)	15 GPR × 32 bits	31 GPR × 32 bits
I/O	Memory mapped	Memory mapped

Daten – Transferbefehle:

	Instruction name	ARM	MIPS
Register-register	Add	add	addu, addiu
	Add (trap if overflow)	adds; swivs	add
	Subtract	sub	subu
	Subtract (trap if overflow)	subs; swivs	sub
	Multiply	mul	mult, multu
	Divide	—	div, divu
	And	and	and
	Or	orr	or
	Xor	eor	xor
	Load high part register	—	lui
	Shift left logical	lsl ¹	sllv, sll
	Shift right logical	lsr ¹	srlv, srl
	Shift right arithmetic	asr ¹	srav, sra
	Compare	cmp, cmn, tst, teq	slt/i,slt/iu
Data transfer	Load byte signed	ldrsb	lb
	Load byte unsigned	ldrb	lbu
	Load halfword signed	ldrsh	lh
	Load halfword unsigned	ldrh	lhu
	Load word	ldr	lw
	Store byte	strb	sb
	Store halfword	strh	sh
	Store word	str	sw
	Read, write special registers	mrs, msr	move
	Atomic Exchange	swp, swpb	ll;sc

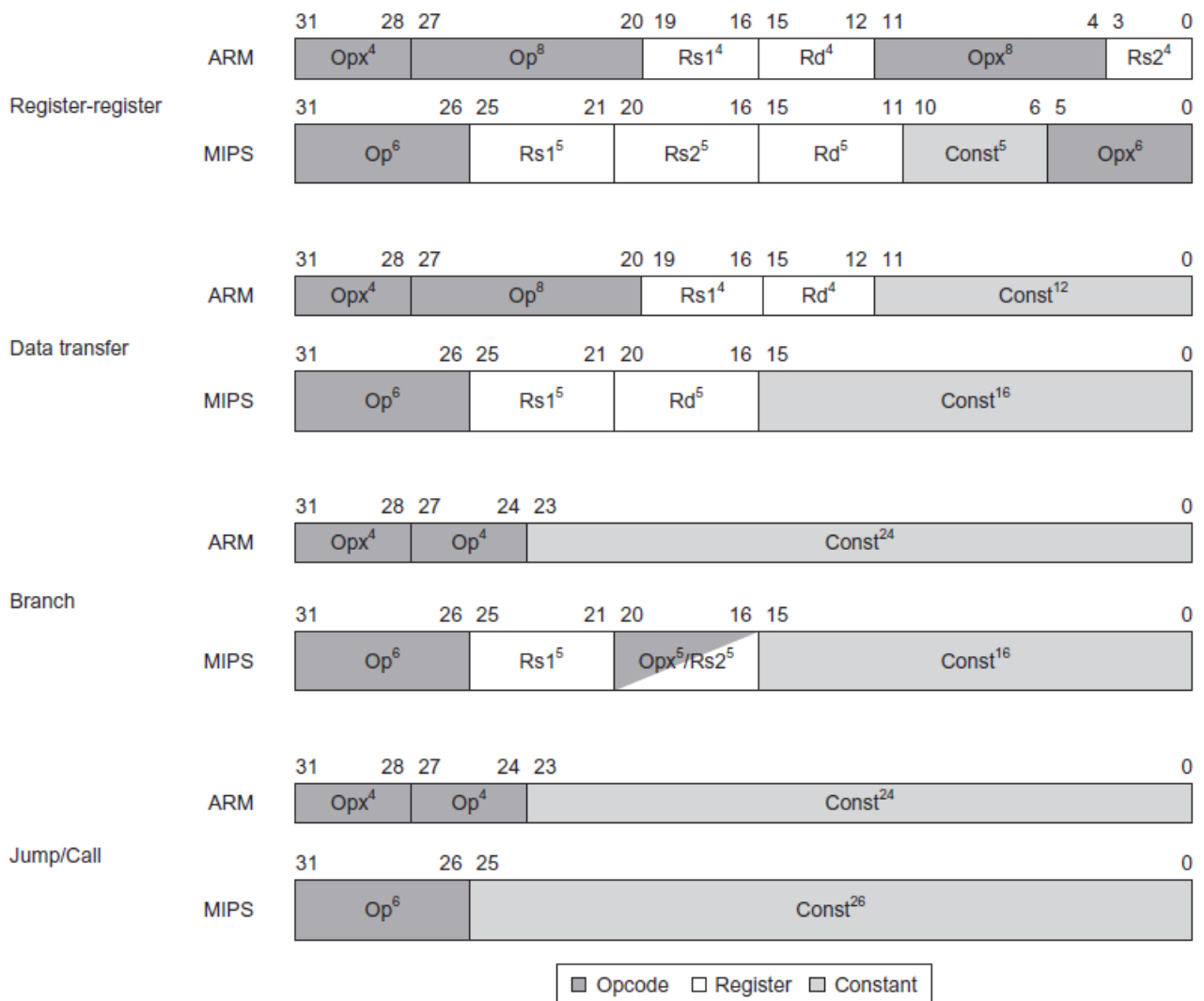
Adressierung

Addressing mode	ARM	MIPS
Register operand	X	X
Immediate operand	X	X
Register + offset (displacement or based)	X	X
Register + register (indexed)	X	—
Register + scaled register (scaled)	X	—
Register + offset and update register	X	—
Register + register and update register	X	—
Autoincrement, autodecrement	X	—
PC-relative data	X	—

Unterschiede

Name	Definition	ARM	MIPS
Load immediate	$Rd = Imm$	mov	addi \$0,
Not	$Rd = \sim(Rs1)$	mvn	nor \$0,
Move	$Rd = Rs1$	mov	or \$0,
Rotate right	$Rd = Rs \ll i$ $Rd_{0 \dots i-1} = Rs_{31-i \dots 31}$	ror	
And not	$Rd = Rs1 \& \sim(Rs2)$	bic	
Reverse subtract	$Rd = Rs2 - Rs1$	rsb, rsc	
Support for multiword integer add	CarryOut, $Rd = Rd + Rs1 + OldCarryOut$	adcs	—
Support for multiword integer sub	CarryOut, $Rd = Rd - Rs1 + OldCarryOut$	sbc	—

Instruktionssatzformate



8.4 Implementierung eines RISC-Prozessors

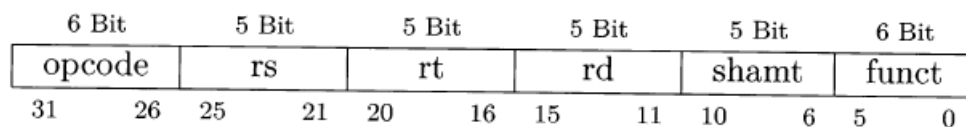
8.4.1 Registersatz MIPS:

Nummer	Name	Bedeutung
0	\$zero	Mit dem konstanten Wert 0 belegt
1	\$at	Für Pseudooperationen reserviert
2-3	\$v0-\$v1	Enthält den Rückgabewert von Funktionen
4-7	\$a0-\$a3	Für die Argumente von Funktionsaufrufen
8-15	\$t0-\$t7	Ungesicherte Register
16-23	\$s0-\$s7	Gesicherte Arbeitsregister
24-25	\$t8-\$t9	Ungesicherte Register
26-27	\$k0-\$k1	Für das Betriebssystem reserviert.
28	\$gp	Zeiger auf globale Variablen
29	\$sp	Stackpointer/Stapelzeiger
30	\$fp	Zeiger auf Aufruf-Rahmen
31	\$ra	Rücksprungadresse, wird vom Befehl jal gesetzt

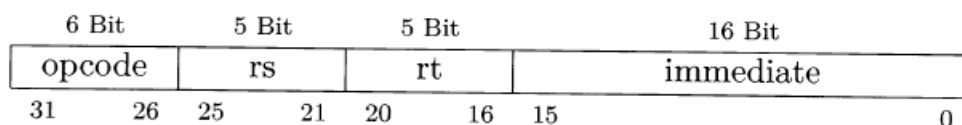
8.4.2 Befehlsformate:

Dem vierten Entwurfsprinzip folgend kennt die MIPS-Architektur insgesamt drei verschiedene Befehlsformate:

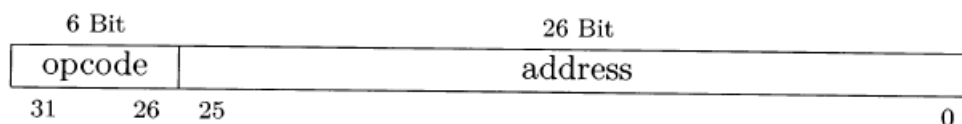
1. R-Typ: hier kommen ausnahmslos Registeroperanden zum Einsatz



2. I-Typ: Mischung aus Registeroperanden und einem Direktwert (Immediate)



3. J-Typ: Spezielles Format für unbedingte Programmsprünge



Jeder Befehl besteht aus nur einem 32 Bit breiten Wort. Durch diese feste Distanz entsteht ein 4 Byte-Alignment. Bei allen drei Formaten wird der Opcode in den höchstwertigen sechs Bit des Befehlswortes übergeben.

Im Format R (Register) haben alle Befehle den Opcode 000000. Um welchen Befehl es sich genau handelt, wird im ebenfalls sechs Bit großen Bereich funct übergeben. Neben der Angabe des Ziel- (rd) und der zwei Quellregister (rs und rt) gibt es noch ein Feld shamt (shift amount, optionales Schieben) zur Angabe der Schritte bei Schiebefehlen.

Befehle im Format I enthalten Angaben zum Ziel- (rt) und einem Quellregister (rs), sowie eine 16 Bit

breite Konstante (Immediate). Die Übergabe eines 32 Bit breiten konstanten Wertes ist prinzipbedingt nicht möglich. Hier muss sich ein Compiler bzw. Assemblerprogrammierer mit einer kleinen Programmsequenz behelfen.

Den einfachsten Aufbau haben die **Befehle im Format J** (Jump, Sprung). Durch das schon angesprochene 4Byte-Alignment müssen bei Sprungbefehlen die Adressen nicht vollständig, sondern nur von Bit 2 beginnend im Opcode abgelegt werden. Damit lassen sich viermal größere Bereiche adressieren, als Adressbits zur Verfügung stehen. Zum Beispiel verwenden die unbedingten Sprungbefehle (ja, jal) eine absolute Adressierung unter Angabe einer 26 Bit Adresse im Opcode dieser Befehle. Durch die Erweiterung können Adressen im Bereich von 0 bis 2^{28} Bit (0...256 MByte) angesprochen werden. Selbst darüber liegende Adressen könnten diese Sprungbefehle erreichen, da die oberen vier Bits der neuen Adresse nicht etwa mit Nullen aufgefüllt, sondern vom alten Inhalt des Befehlszählers übernommen werden. Da ein so großer Adressraum für den geplanten Prozessor nicht vorgesehen ist, wird zur Vereinfachung des eigenen Designs nachfolgend auf diese Übernahme der höchstwertigen vier Bits in die Adresse verzichtet.

Die im I-Format realisierten bedingten relativen Sprünge (beq, bne, ...) profitieren ebenfalls von der Ausrichtung des Speichers auf jeweils vier Byte große Blöcke. Sie können trotz des im Opcode für die Angabe der Konstante verfügbaren Bereichs von nur 16 Bit den viermal so großen Adressraum von +/- 128K adressieren. Alle arithmetischen und logischen Befehle eines MIPS-Prozessors arbeiten nur mit Registeroperanden. Da eine große Registerbank vorhanden ist, können so ganze Programmteile ohne Zuriff. auf den Datenspeicher auskommen. Erklärtes Ziel ist es, langsame Speicherzugriffe unbedingt zu vermeiden. Der Zugriff auf den Datenspeicher erfolgt über spezielle Datentransferbefehle (lw, sw). Programm- und Datenspeicher sind getrennt (Harvard-Architektur).

Der zu implementierende Befehlssatz beschränkt sich (entsprechend dem dritten Entwurfsprinzip) nicht nur auf die unbedingt notwendigen bzw. nur unter großen Anstrengungen zu ersetzenden Befehle des MIPS, sondern versucht zusätzlich die am häufigsten eingesetzten Operationen abzubilden. Zum Beispiel wäre die Implementierung der Additionsbefehle mit Konstante (addi, addiu) nicht unbedingt nötig. Doch gerade diese Befehle kommen sehr oft zum Einsatz. Auch würde ein bedingter Sprungbefehl, welcher auf Gleichheit zweier Registerinhalte prüft, genügen. Durch das Bereitstellen von zwei bedingten Sprüngen mit gegensätzlicher Sprungbedingung (beq, bne) sind aber wesentlich besser lesbare Programme möglich.

Der Aufruf von Unterprogrammen erfolgt beim MIPS über einen speziellen Sprungbefehl (jal, jump and link). Dieser sichert die Adresse des nächsten abzuarbeitenden Befehls im Register \$ra (return address, Register 31) und verzweigt dann zur angegebenen Adresse. Für die Übergabe von Argumenten sind die vier Register \$a0 bis \$a3 und für den Rückgabewert die Register \$v0 und \$v1 reserviert. Der Rücksprung erfolgt über jr \$ra jump register return address). Ein spezieller Return-Befehl ist deshalb nicht nötig.

8.4.3 Überlegungen zur Implementierung

Durch die ausführliche Dokumentation und das (aufgrund des konsequenten Einsatzes der RISC-Grundprinzipien) für einen modernen Prozessor relativ einfach gehaltene Design eignet sich das MIPS-Konzept sehr gut für den Einsatz in der Lehre. Dies umfasst das weite Feld von der Programmierung auf Maschinensprachniveau bis hin zum Entwurf des Prozessors in einer Hardwarebeschreibungssprache.

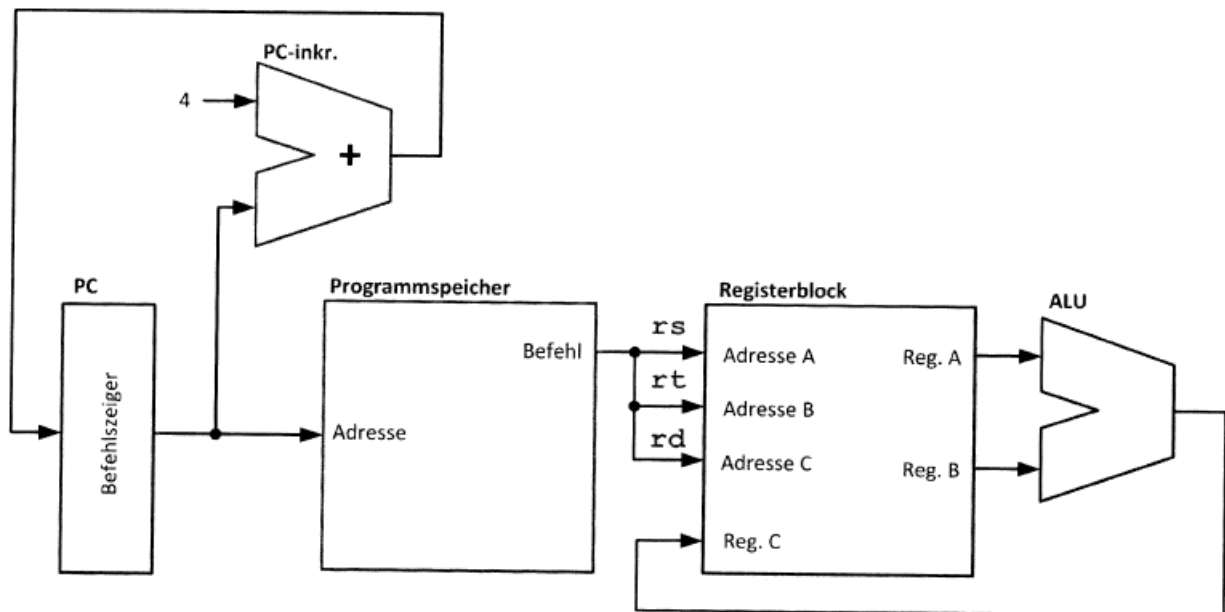
Aus diesem Grund gibt es für diesen Prozessorkern eine Vielzahl von Implementierungen in den unterschiedlichsten Hardwarebeschreibungssprachen. Neben diversen Hochschulprojekten gibt es mehrere interessante Projekte auf der Webseite von OpenCores.

Die Grundlage der meisten Implementierungen ist das Buch „Computer organization Design“ von Petterson und Hennessy. In dieser Abhandlung wird zwar keine konkrete Umsetzung des kompletten Prozessors in eine Hardwarebeschreibungssprache beschrieben, dafür aber alle notwendigen Vorüberlegungen und Konzepte erläutert.

8.4.4 Teilmenge der implementierten Befehle:

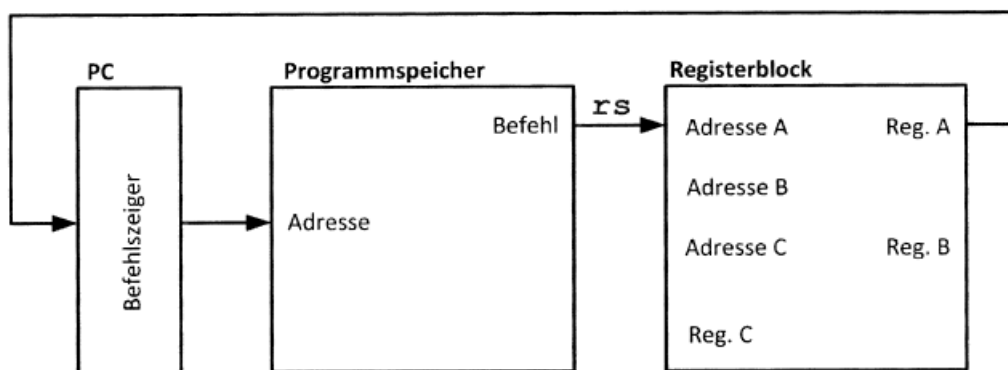
Mnemonic	Typ	Name	Beschreibung
Arithmetische Befehle			
ADD rd,rs,rt	R	Add Signed	$R[rd] := R[rs] + R[rt]$
ADDI rd,rs,Im16	I	Add Signed	$R[rd] := R[rs] + Im16$
ADDU rd,rs,rt	R	Add Unsigned	$R[rd] := R[rs] + R[rt]$
ADDIU rd,rs,Im16	I	Add Unsigned	$R[rd] := R[rs] + Im16$
SUB rd,rs,rt	R	Subtract Signed	$R[rd] := R[rs] - R[rt]$
SUBU rd,rs,rt	R	Subtract Unsigned	$R[rd] := R[rs] - R[rt]$
SLL rd,rs,shamt	R	Shift Left Logical	$R[rd] := R[rs] \ll shamt$
SRL rd,rs,shamt	R	Shift Right Logical	$R[rd] := R[rs] \gg shamt$
SLT rd,rs,rt	R	Set Less Than	if ($R[rs] < R[rt]$) $R[rd] := 1$ else 0
SLTU rd,rs,rt	R	Set Less Than Unsig.	if ($R[rs] < R[rt]$) $R[rd] := 1$ else 0
Logische Befehle			
AND rd,rs,rt	R	AND	$R[rd] := R[rs] \text{ AND } R[rt]$
ANDI rd,rs,Im16	I	AND Immediate	$R[rd] := R[rs] \text{ AND } Im16$
OR rd,rs,rt	R	OR	$R[rd] := R[rs] \text{ OR } R[rt]$
ORI rd,rs,Im16	I	OR Immediate	$R[rd] := R[rs] \text{ OR } Im16$
NOR rd,rs,rt	R	NOR	$R[rd] := R[rs] \text{ NOR } R[rt]$
XOR rd,rs,rt	R	XOR	$R[rd] := R[rs] \text{ XOR } R[rt]$
Verzweigungsbefehle			
BEQ rs,rt,Im16	I	Branch On Equal	if ($R[rs] = R[rt]$) $PC := PC + 4 + Im16$
BNE rs,rt,Im16	I	Branch On Not Equal	if ($R[rs] \neq R[rt]$) $PC := PC + 4 + Im16$
Unbedingte Sprungbefehle			
J address	J	Jump	$PC := address$
JR rs	R	Jump Register	$PC := R[rs]$
JAL address	J	Jump And Link	$\$31 := PC + 4; PC := address$
Datentransferbefehle			
LW rt,Im16(rs)	I	Load Word	$R[rt] := Mem[R[rs] + Im16]$
SW rt,Im16(rs)	I	Store Word	$Mem[R[rs] + Im16] := R[rt]$

8.4.5 CPU Version 0: R-Befehle

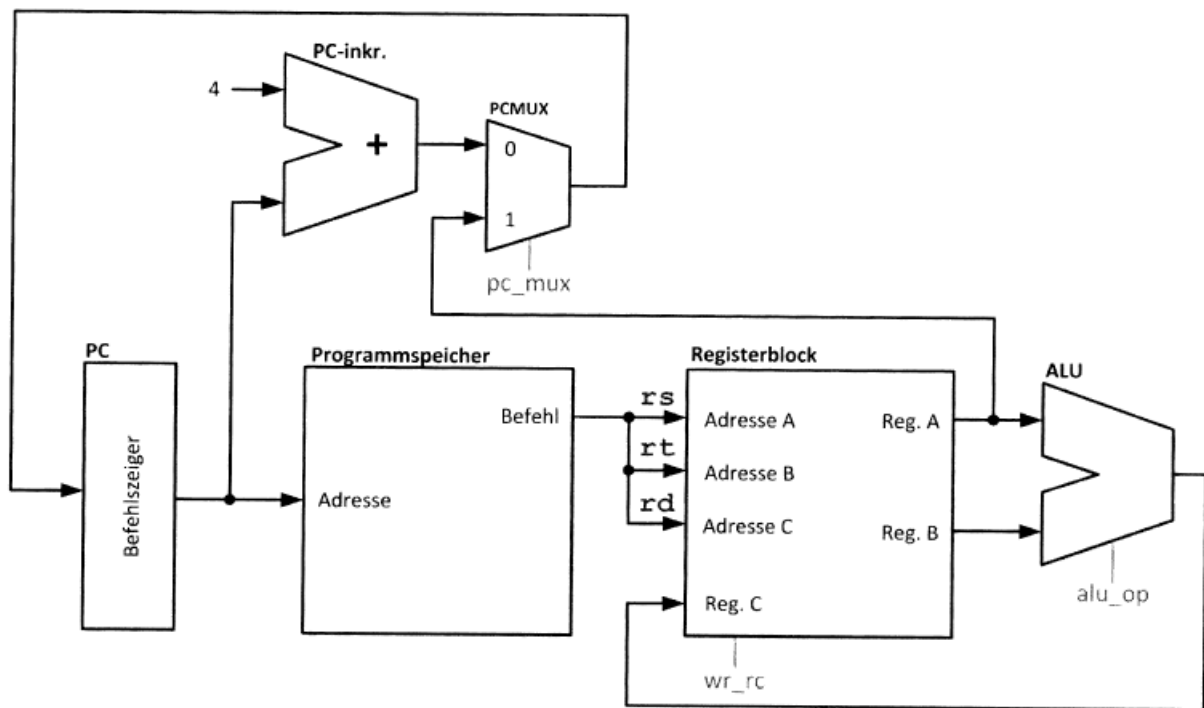


8.4.6 CPU Version 1: J-Befehl

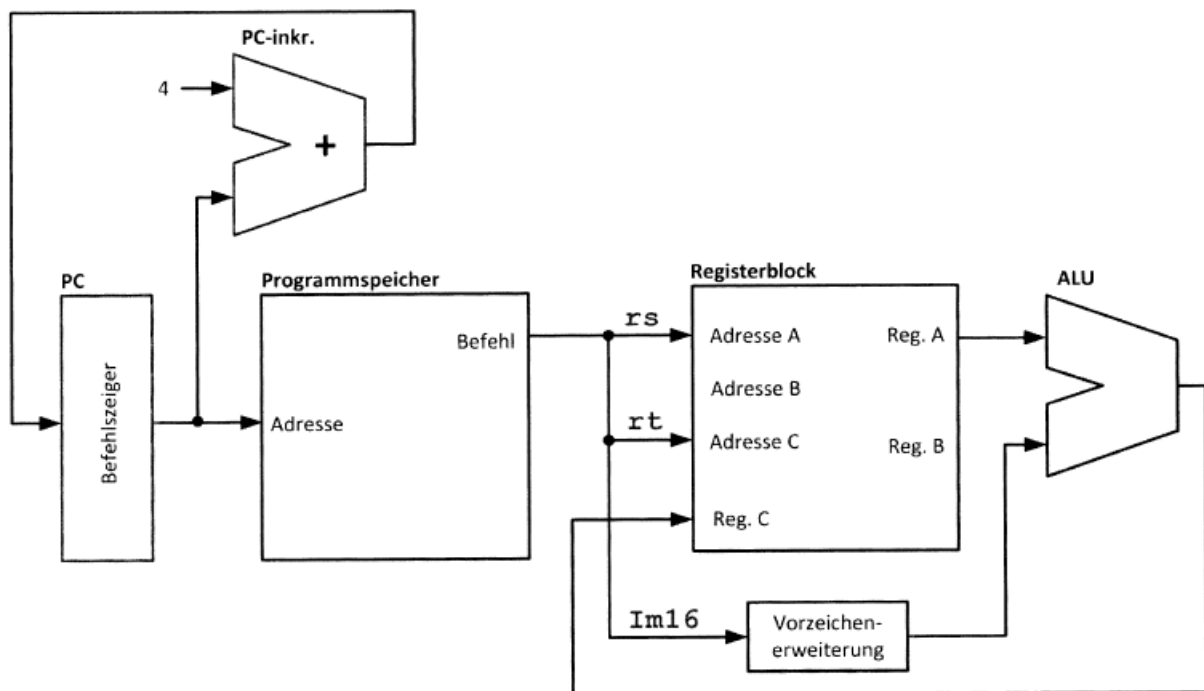
Datenpfad Jump



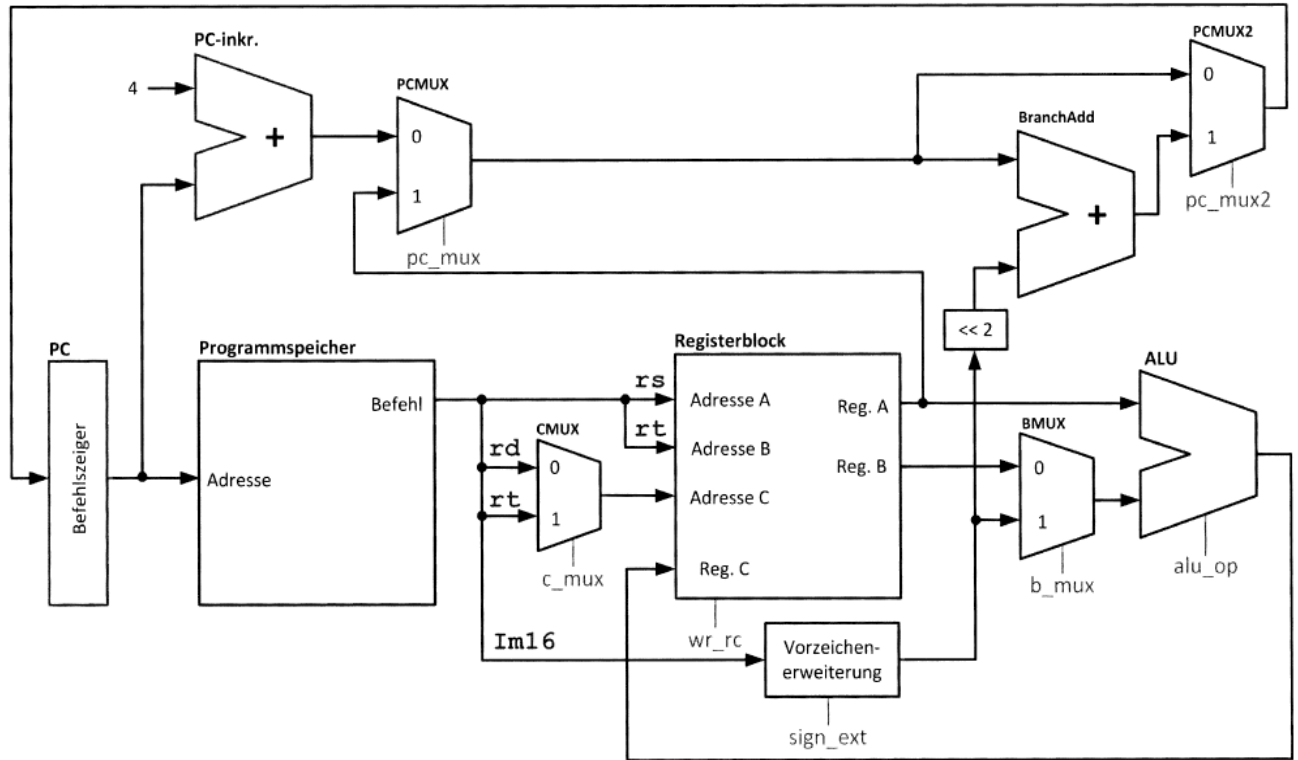
CPU-Version1



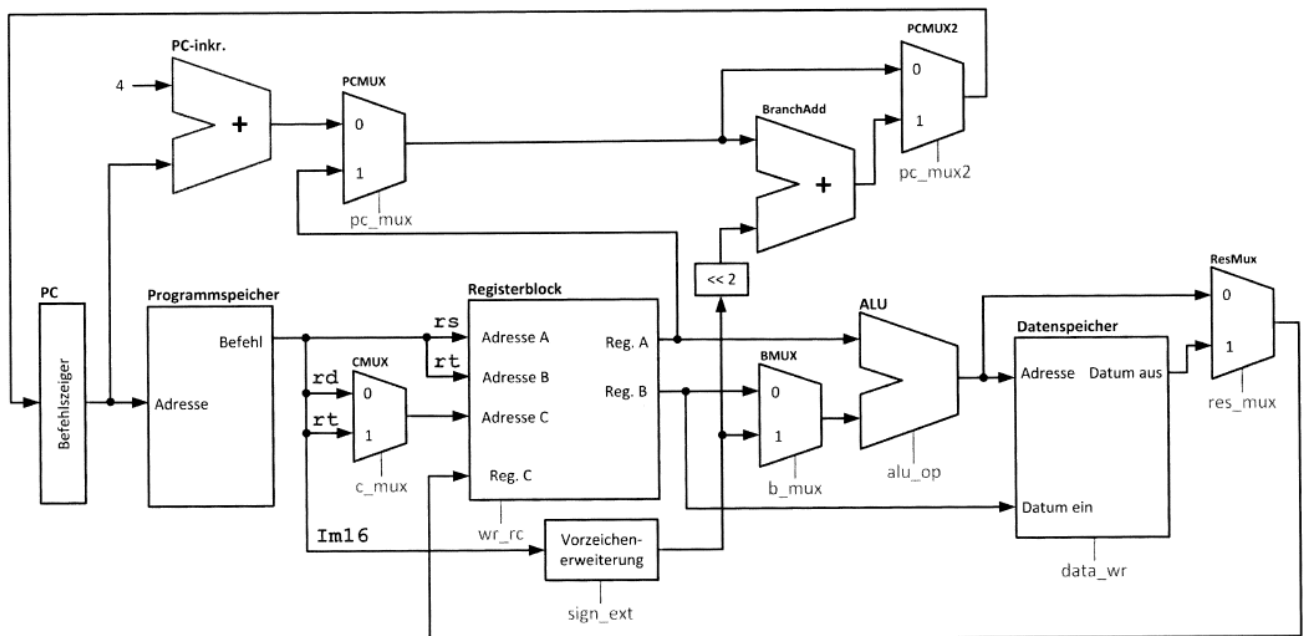
8.4.7 CPU Version 2: Immediate



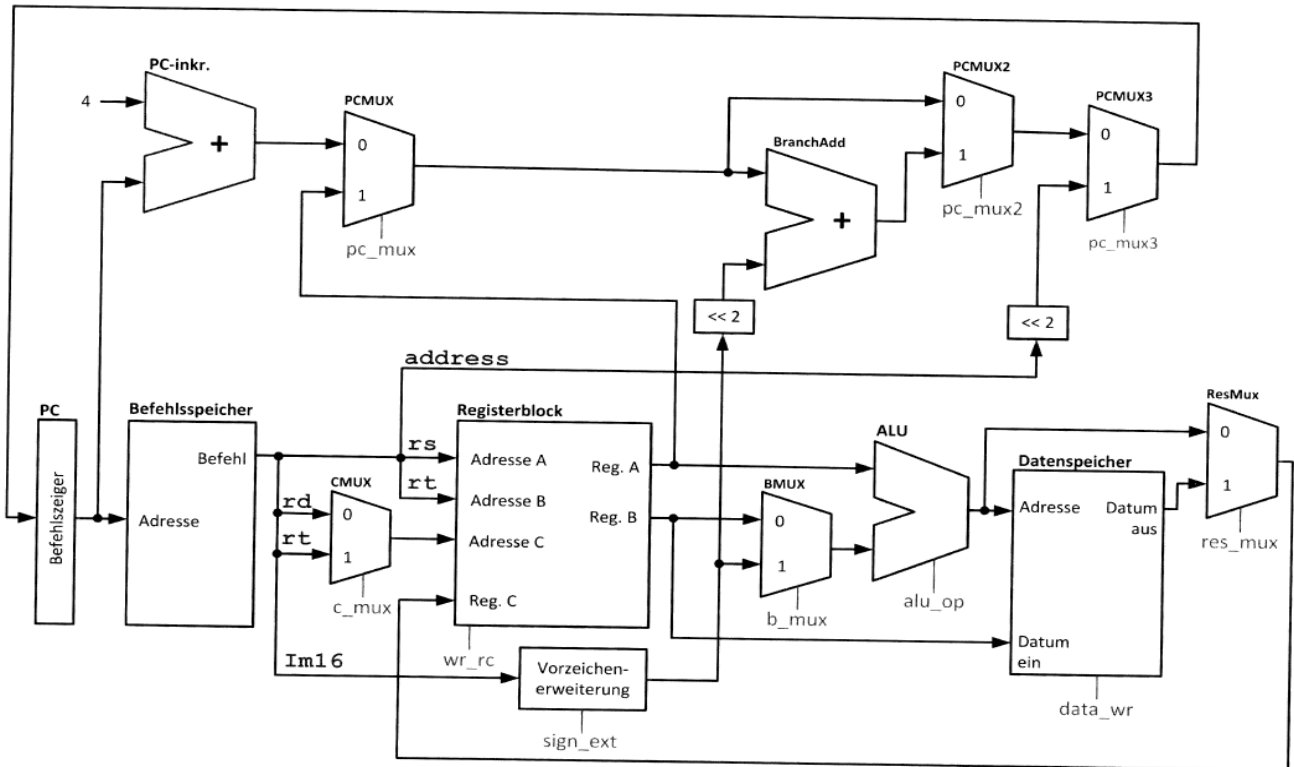
8.4.8 CPU Version 3: Verzweigung



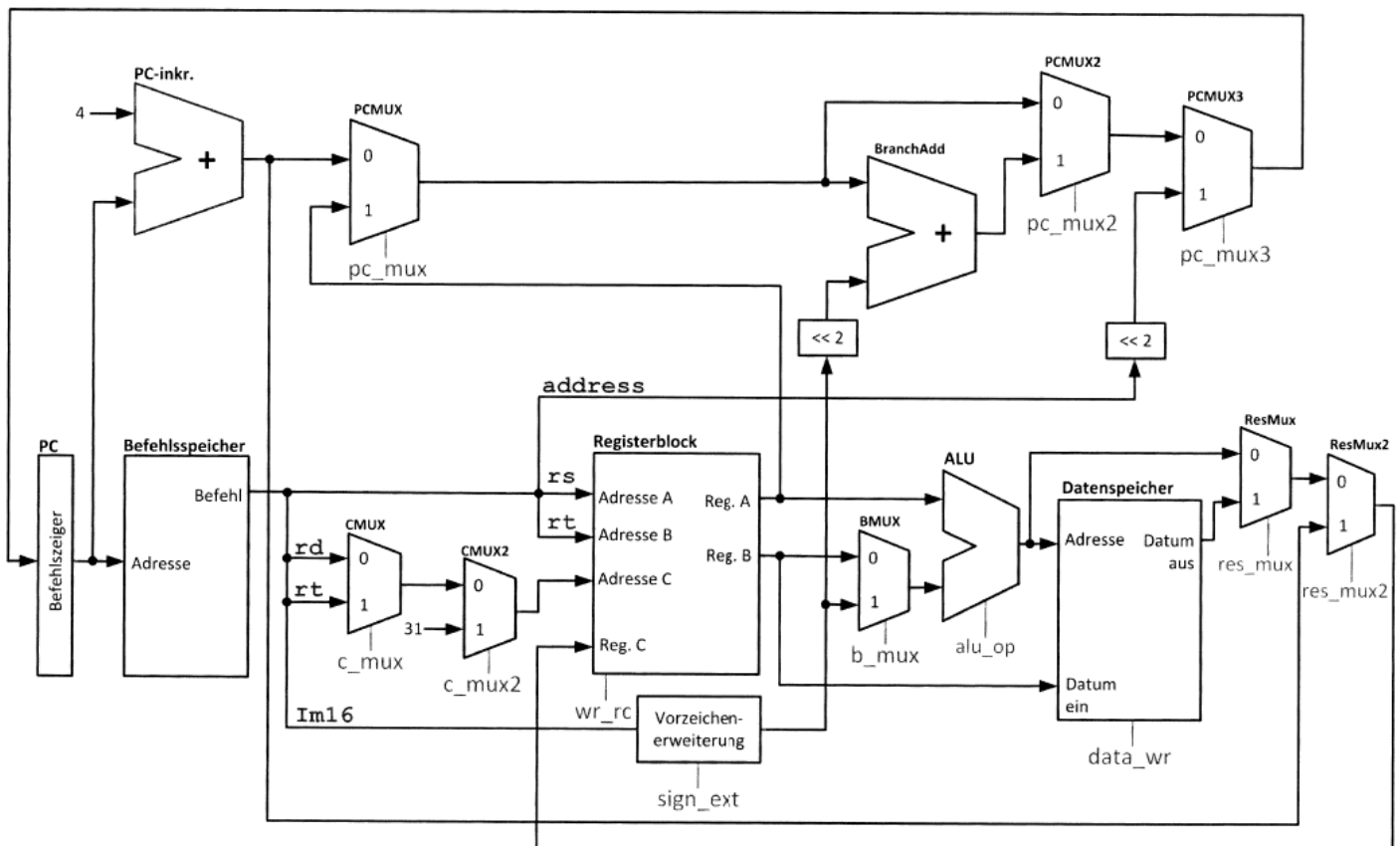
8.4.9 CPU Version 4: Load-Store



8.4.10 CPU Version 5: Sprung



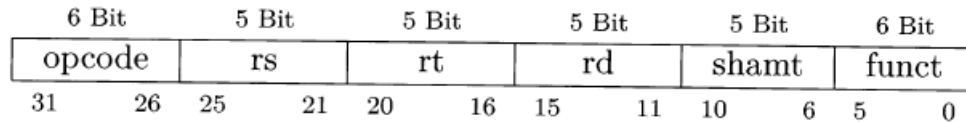
8.4.11 CPU Version 6: Jump and Link



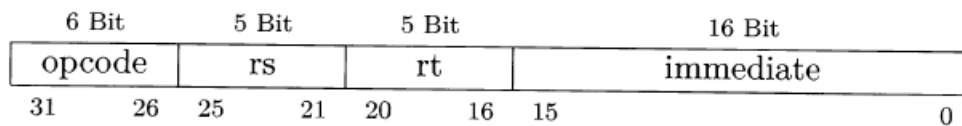
8.5 HTL-MIPS

8.5.1 Befehlsformate:

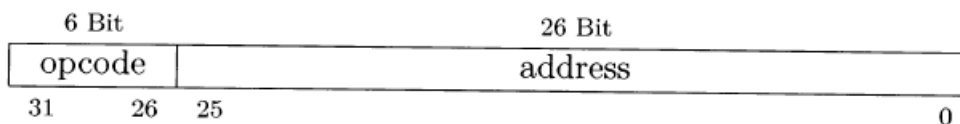
1. R-Typ: hier kommen ausnahmslos Registeroperanden zum Einsatz



2. I-Typ: Mischung aus Registeroperanden und einem Direktwert (Immediate)



3. J-Typ: Spezielles Format für unbedingte Programmsprünge



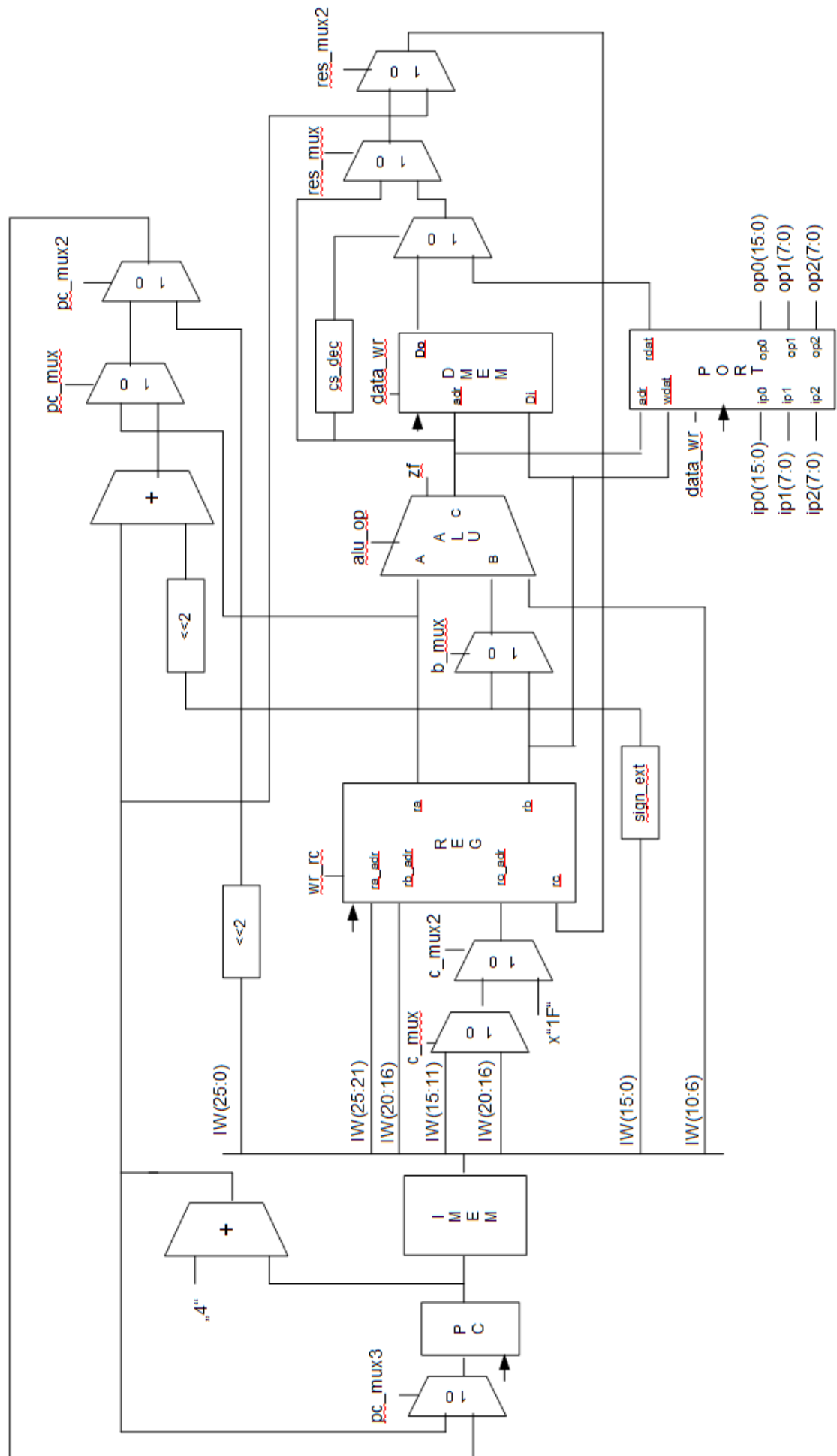
8.5.2 Registersatz

Nummer	Name	Bedeutung
0	\$zero	Mit dem konstanten Wert 0 belegt
1	\$at	Für Pseudooperationen reserviert
2-3	\$v0-\$v1	Enthält den Rückgabewert von Funktionen
4-7	\$a0-\$a3	Für die Argumente von Funktionsaufrufen
8-15	\$t0-\$t7	Ungesicherte Register
16-23	\$s0-\$s7	Gesicherte Arbeitsregister
24-25	\$t8-\$t9	Ungesicherte Register
26-27	\$k0-\$k1	Für das Betriebssystem reserviert.
28	\$gp	Zeiger auf globale Variablen
29	\$sp	Stackpointer/Stapelzeiger
30	\$fp	Zeiger auf Aufruf-Rahmen
31	\$ra	Rücksprungadresse, wird vom Befehl jal gesetzt

8.5.3 MIPS-Assembler-Subset CPU6_15

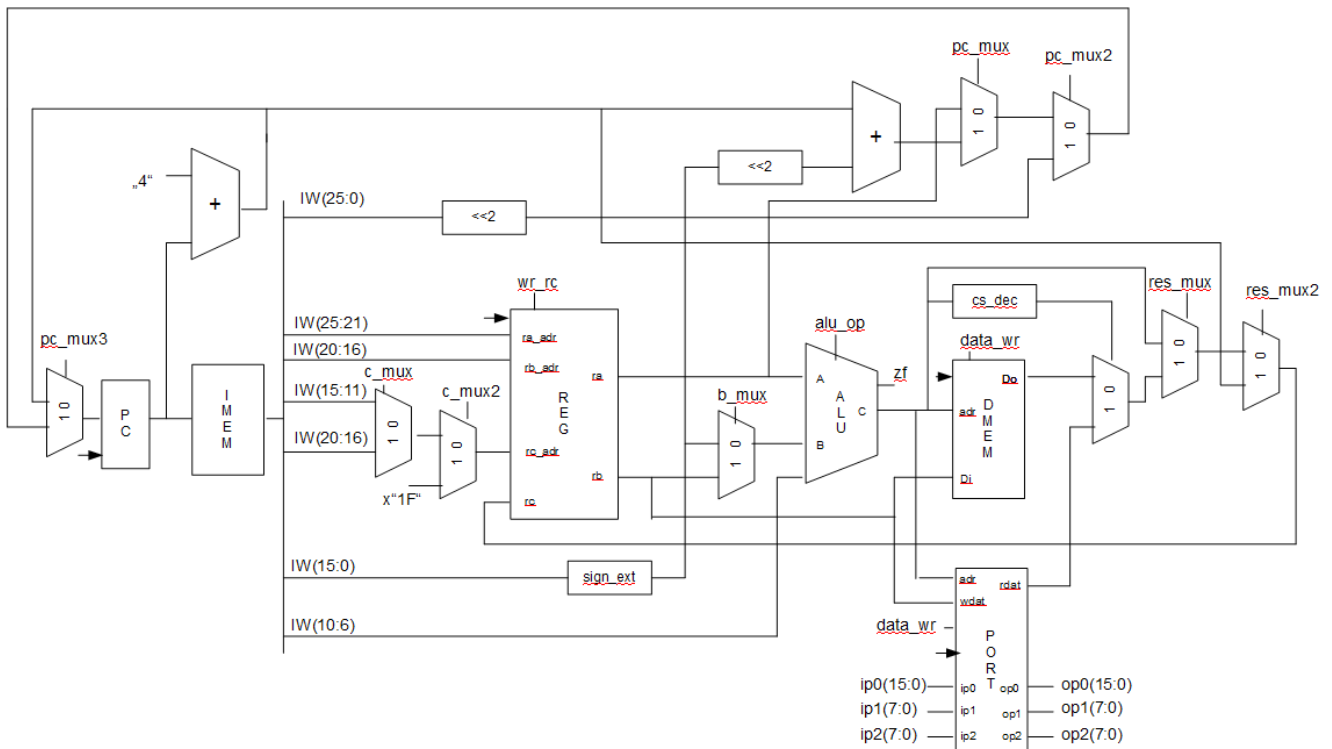
Mnemonik	Typ	Name	Beschreibung
Arithmetische Befehle			
ADD rd, rs, rt	R	Add Signed	$R[rd] := R[rs] + R[rt]$
ADDI rt, rs, Im16	I	Add Signed Immediate	$R[rt] := R[rs] + Im16$
ADDU rd, rs, rt	R	Add Unsigned	$R[rd] := R[rs] + R[rt]$
ADDIU rt, rs, Im16	I	Add Unsigned Immediate	$R[rt] := R[rs] + Im16$
SUB rd, rs, rt	R	Subtract Signed	$R[rd] := R[rs] - R[rt]$
SUBU rd, rs, rt	R	Subtract Unsigned	$R[rd] := R[rs] - R[rt]$
SLL rd, rs, shamt	R	Shift Left Logical	$R[rd] := R[rs] \ll shamt$
SRL rd, rs, shamt	R	Shift Right Logical	$R[rd] := R[rs] \gg shamt$
SLT rd, rs, rt	R	Set Less Than	If ($R[rs] < R[rt]$) $R[rd] := 1$ else 0
SLTU rd, rs, rt	R	Set Less Than Unsigned	If ($R[rs] < R[rt]$) $R[rd] := 1$ else 0
Logische Befehle			
AND rd, rs, rt	R	And	$R[rd] := R[rs] \text{ AND } R[rt]$
ADDI rt, rs, Im16	I	And Immediate	$R[rt] := R[rs] \text{ AND } Im16$
OR rd, rs, rt	R	Or	$R[rd] := R[rs] \text{ OR } R[rt]$
ORI rt, rs, Im16	I	Or Immediate	$R[rt] := R[rs] \text{ OR } Im16$
NOR rd, rs, rt	R	Nor	$R[rd] := R[rs] \text{ NOR } R[rt]$
XOR rd, rs, rt	R	Xor	$R[rd] := R[rs] \text{ XOR } R[rt]$
Verzweigungsbefehle			
BEQ rs, rt, Im16	I	Branch on Equal	If ($R[rs] = R[rt]$) $PC := PC + 4 + Im16$
BNE rs, rt, Im16	I	Branch on Not Equal	If ($R[rs] \neq R[rt]$) $PC := PC + 4 + Im16$
Unbedingte Sprungbefehle			
J address	J	Jump	$PC := address$
JR rs	R	Jump Register	$PC := R[rs]$
JAL address	J	Jump And Link	$\$31 := PC + 4; PC := address$
Datentransferbefehle			
LW rt, Im16 (rs)	I	Load Word	$R[rt] := Mem [R[rs] + Im16]$
SW rt, Im16 (rs)	I	Store Word	$Mem [R[rs] + Im16] := R[rt]$

CPU6_15

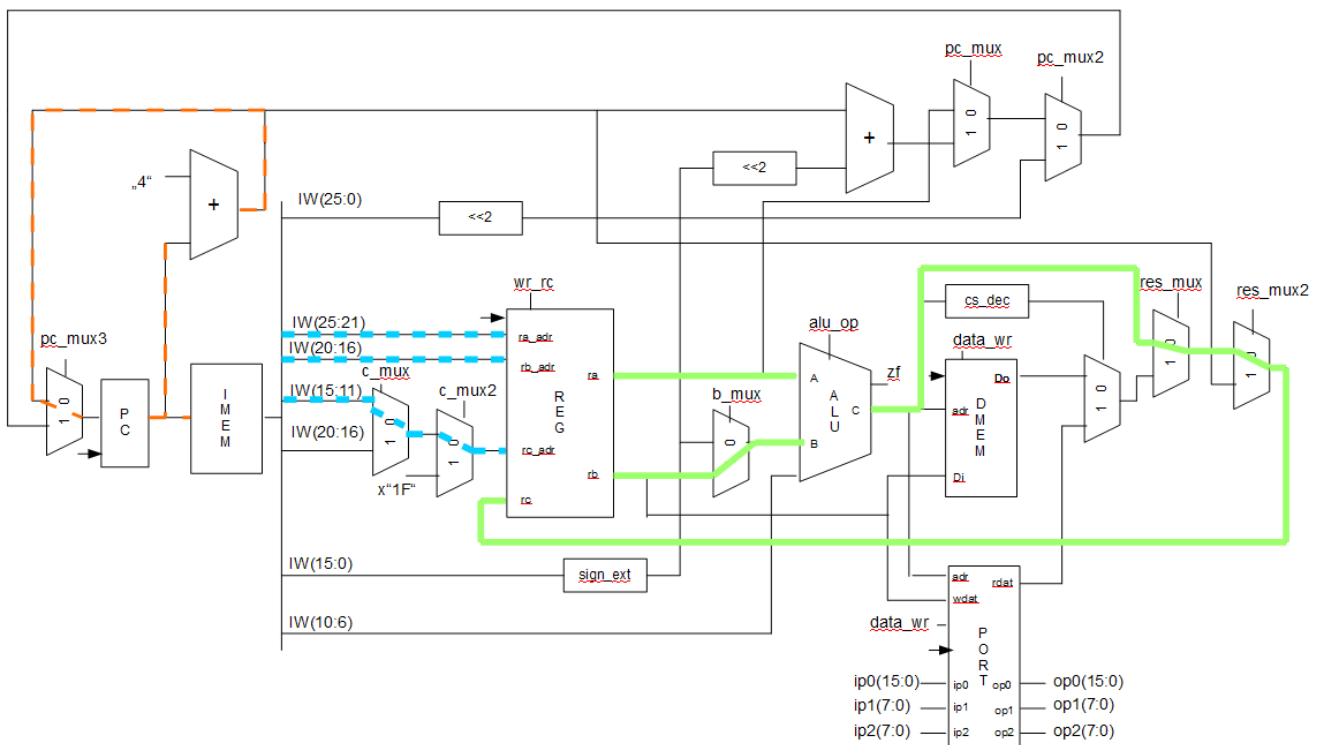


8.5.4 HW-Aufbau CPU6_15

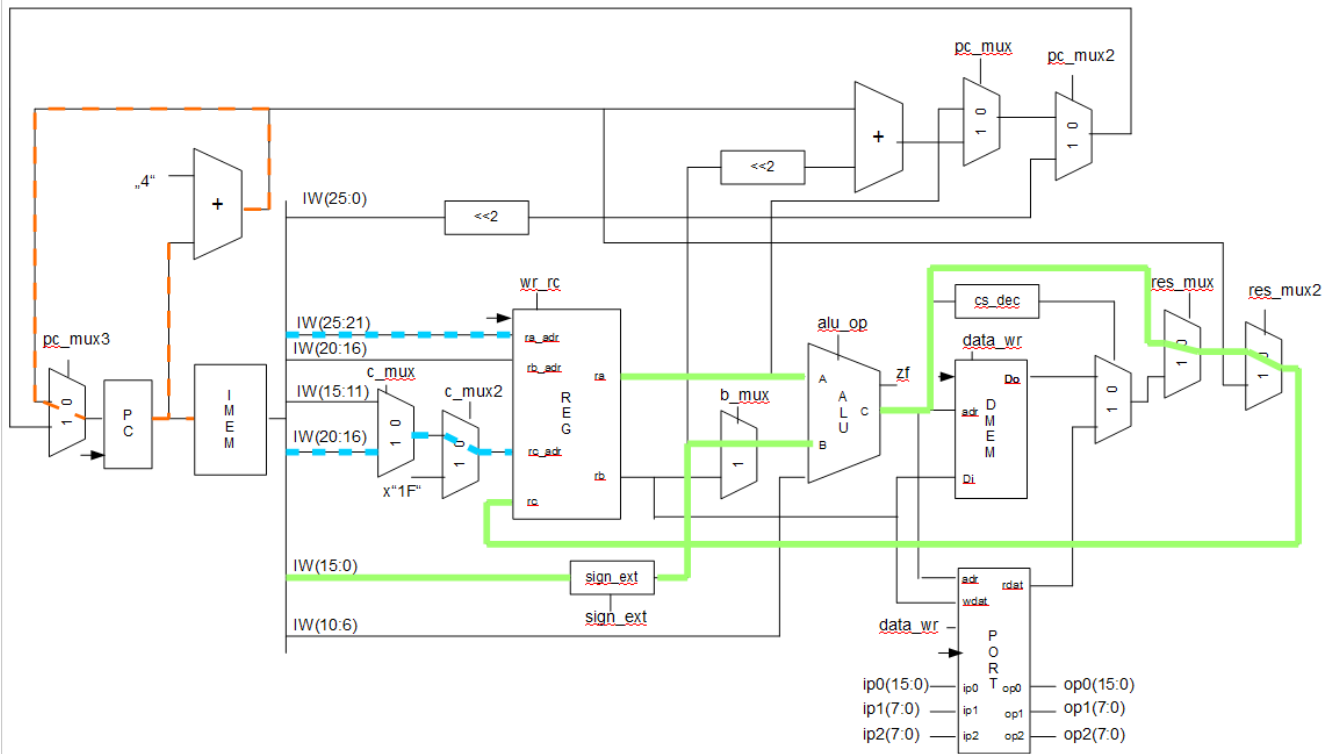
CPU6_15



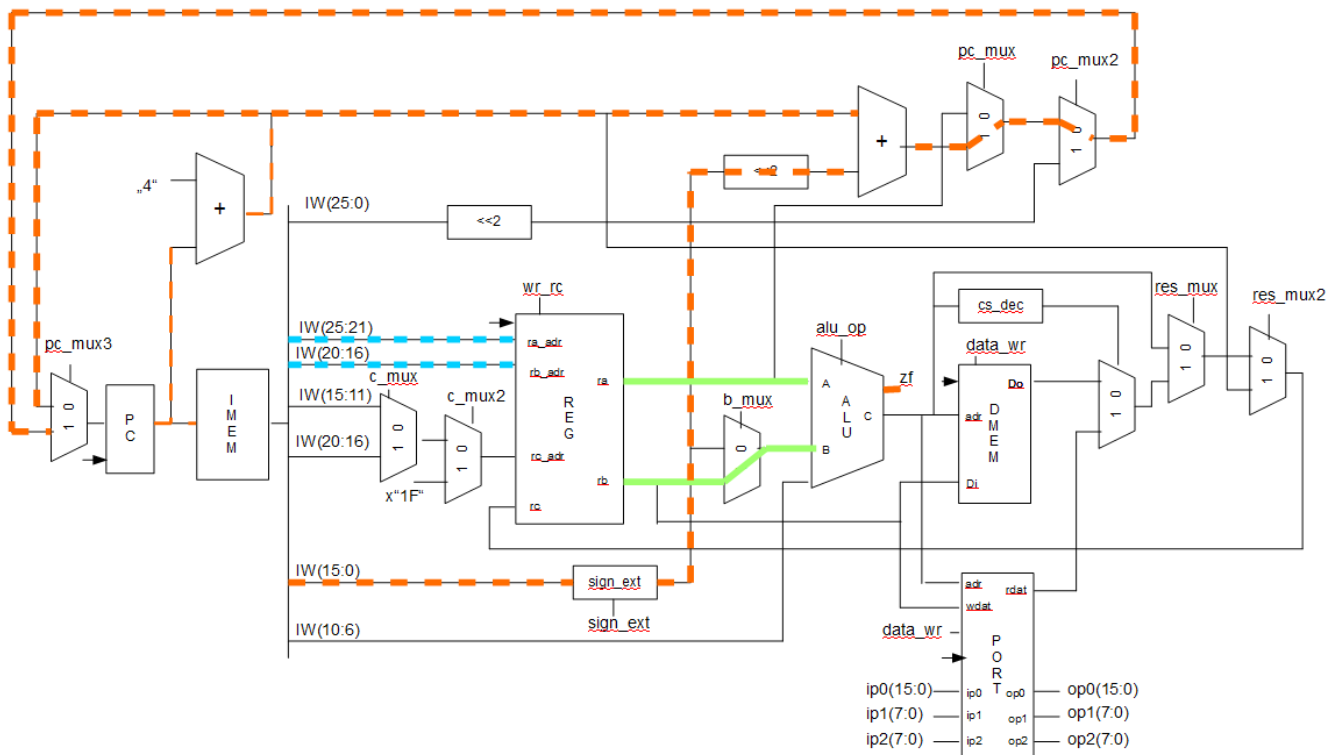
CPU6_15 R-Befehl allg.



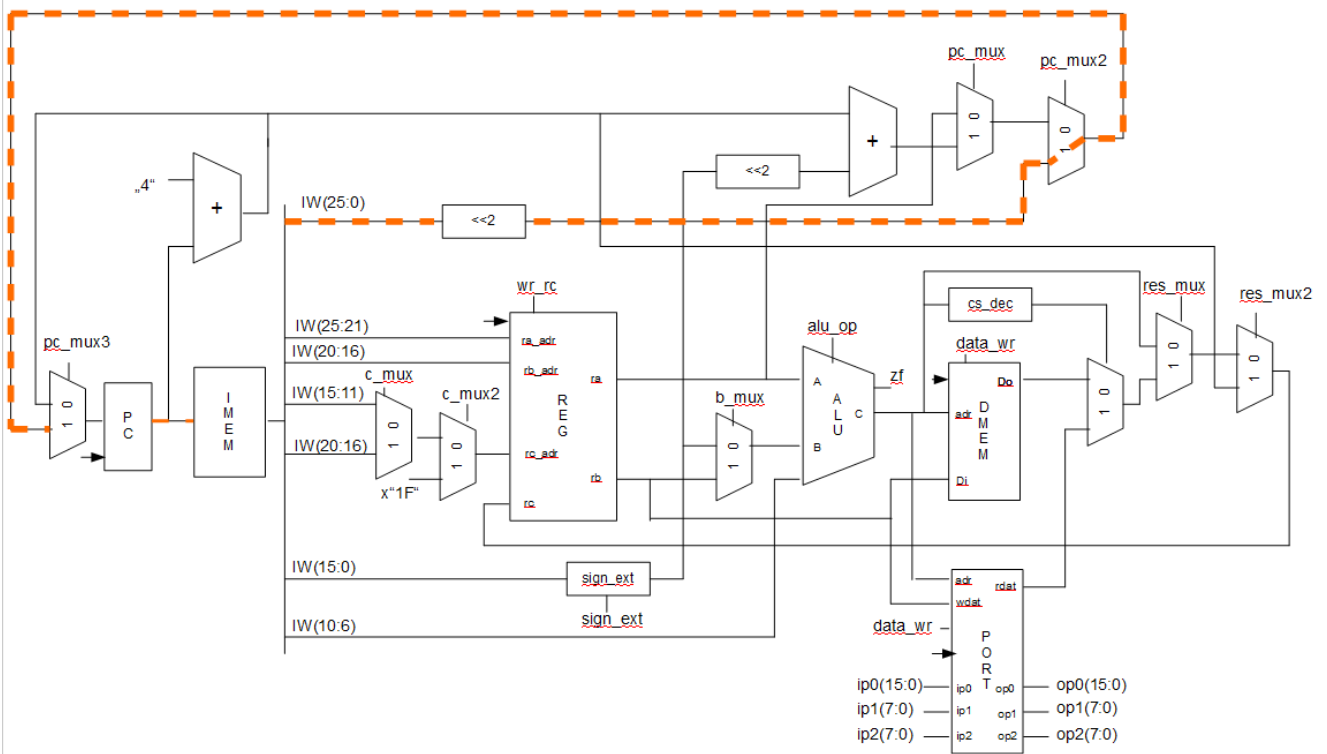
CPU6_15 I-Befehl ANDI



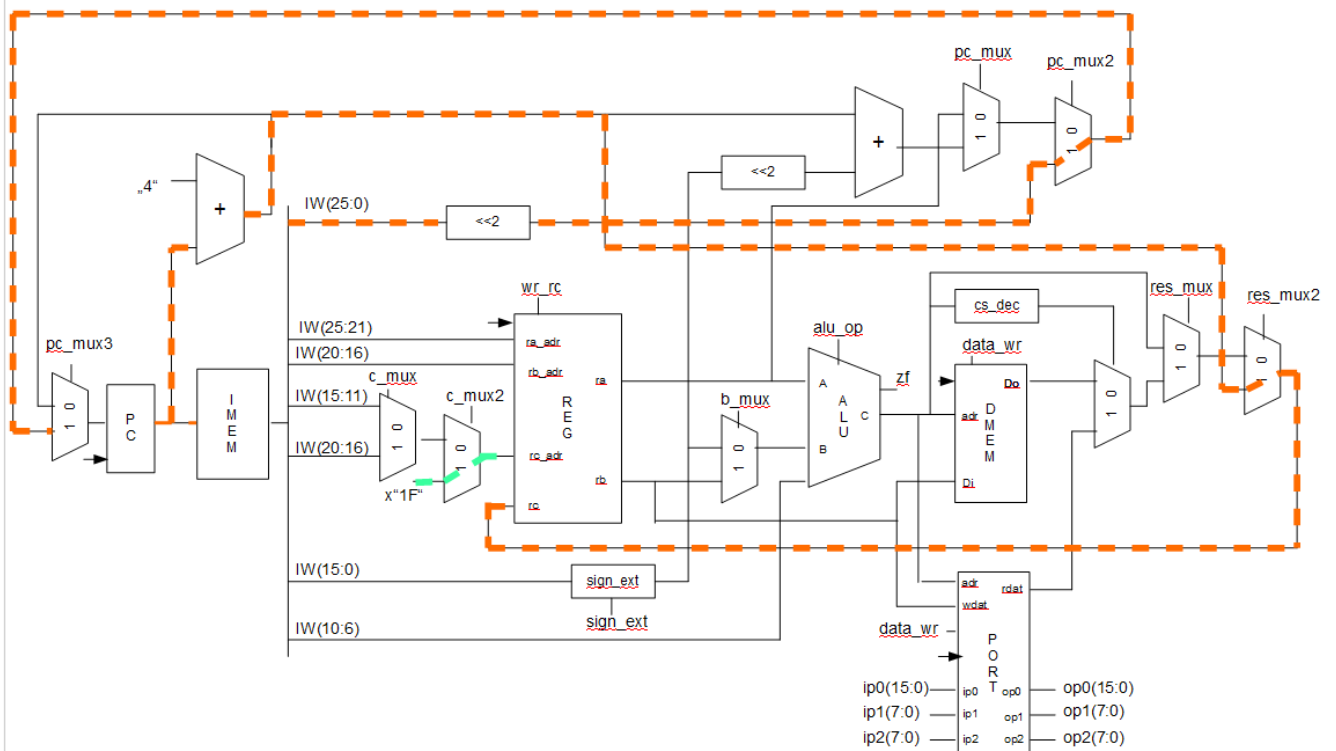
CPU6_15 BEQ



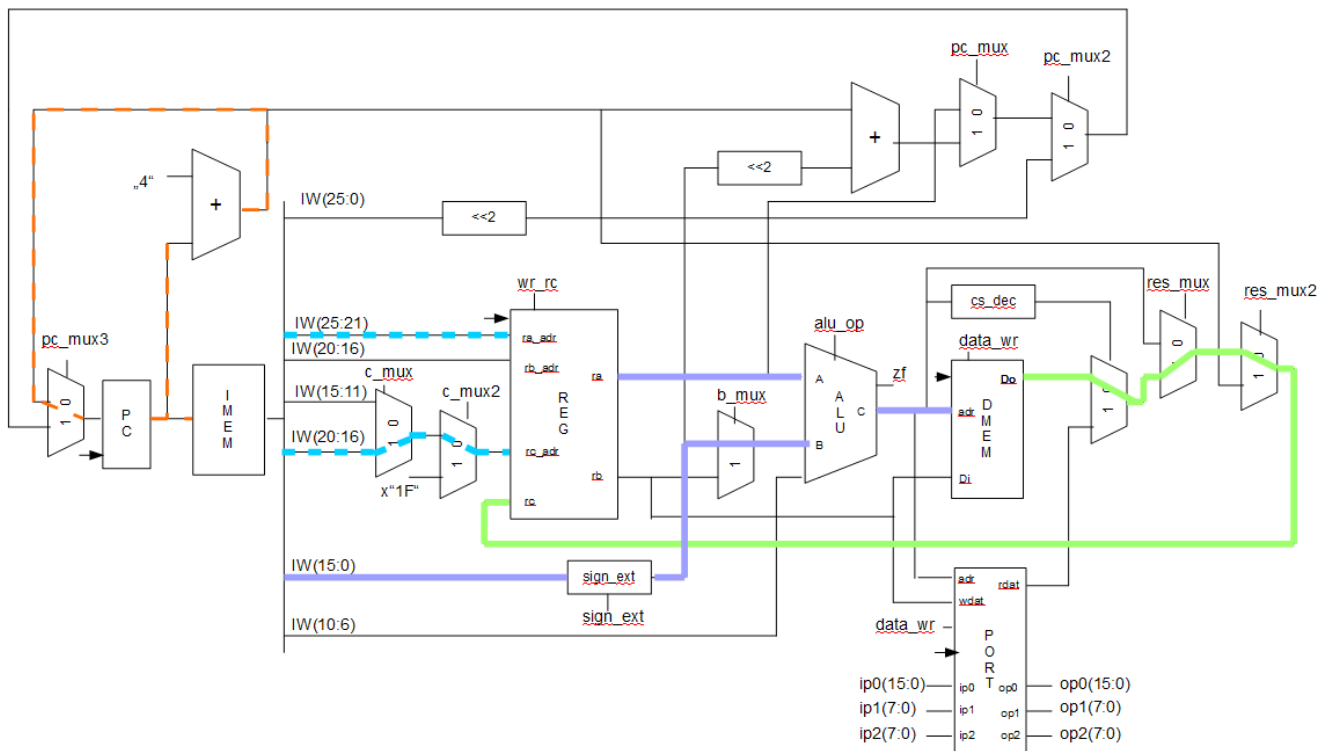
CPU6_15 J-Befehl



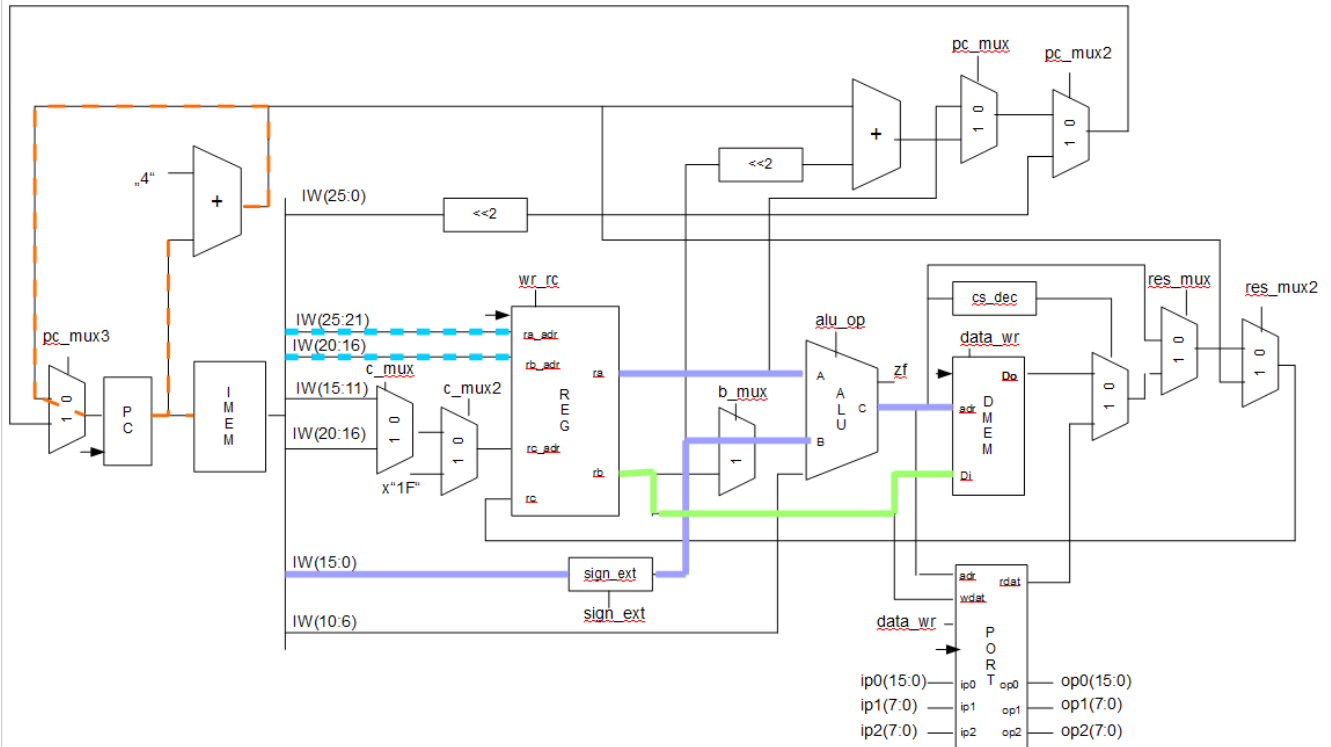
CPU6_15 JAL-Befehl



CPU6_15 LW-Befehl



CPU6_15 SW-Befehl



8.5.5 Programmentwicklung mit MARS

8.5.5.1 Erstes Beispielprogramm:

```
# htmips1.asm
```

```
.eqv iport_1 0x1010
```

```
.eqv oport_1 0x1010
```

```
.text
```

```
M1:  lw $s2, iport_1      # read from input port 1
      addiu $s3,$s2,-1    # modify
      sw $s3, oport_1     # output result
      j M1               # loop
```

Achtung Einstellungen in MARS entsprechend unserer Hardware: (Settings → Memory Configuration)

- Text Base Address 0x“0000“
- Data Base Address 0x“2000“
- Extern Base Address 0x“1000“

richtig setzen (Einstellung: „Compact, Text at \$0“)

Ergebnis von Dump-Memory VHDL-Text-Data

```
constant rom: rom_type := (
```

-- [Adress]	Disassemble	Source
(x"8c121010"),	-- [0x00000000] lw \$18,0x00001010(\$0)	M1: lw \$s2, 0x1010 # read from
input port 1		
(x"2653ffff"),	-- [0x00000004] addiu \$19,\$18,0xffffffff	addiu \$s3,\$s2,-1 # modify
(x"ac121010"),	-- [0x00000008] sw \$18,0x00001010(\$0)	sw \$s2, 0x1010 # output
result		
(x"08000000"),	-- [0x0000000c] j 0x00000000	j M1 # loop

```
-- Den Rest mit NOP (sll $zero, $zero, 0) fuellen
OTHERS => x"0000_0000");
```

8.5.5.2 Übungen mit dem Simulator:

- Single Step
- Breakpoint
- Darstellung des Speicherbereichs „extern“
- Registerinhalte
- Speicher modifizieren
- Tools verwenden:
 - Instruction Counter
 - Instruction Statistics
 - Memory Reference Visualization
 - Screen Magnifier



8.5.5.3 MARS Tutorial

Pete Sanderson, Otterbein College, Ken Vollmar, Missouri State University,

MARS may be downloaded from www.cs.missouristate.edu/MARS.

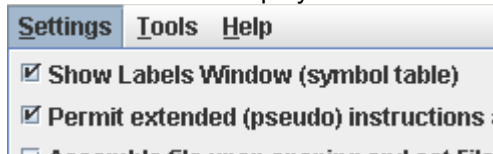
Part 1 : Basic MARS Use

The example program is **Fibonacci.asm** to compute everyone's favorite number sequence.

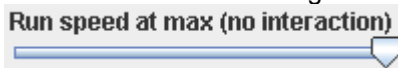
1. Start MARS from the Start menu or desktop icon.
2. Use the menubar File...Open or the Open icon  to open Fibonacci.asm in the default folder. (All icons have menubar equivalents; the remainder of these steps will use the icon whenever possible.)
3. The provided assembly program is complete. Assemble the program using the icon .
4. Identify the location and values of the program's initialized data. Use the checkbox to toggle the display format between decimal and hexadecimal ☐ Hexadecimal Values .
 - The nineteen-element array **fibs** is initialized to zero, at addresses 0x10010000 ... 0x10010048.
 - The data location **size** has value 19_{ten} at 0x1001004c.
 - The addresses 0x10010050 ... 0x1001006c contain null-terminated ASCII strings.

Use the checkbox to toggle the display format between decimal and hexadecimal, ☐ Hexadecimal Values .

5. Use the Settings menu to configure the MARS displays. The settings will be retained for the next MARS session.
 - The Labels display contains the addresses of the assembly code statements with a label, but the default is to *not* show this display. Select the checkbox from the Settings menu.







- Select your preference for allowing pseudo-instructions (programmer-friendly instruction substitutions and shorthand).
 - Select your preference for assembling *only one* file, or *many* files together (all the files in the current folder). This feature is useful for subroutines contained in separate files, etc.
 - Select the startup display format of addresses and values (decimal or hexadecimal).
6. Locate the Registers display, which shows the 32 common MIPS registers. Other tabs in the Registers display show the floating-point registers (Coproc 1) and status codes (Coproc 0).
 7. Use the slider bar to change the run speed to about 10 instructions per second.



This allows us to "watch the action" instead of the assembly program finishing directly.

8. Choose how you will execute the program:

- The  icon runs the program to completion. Using this icon, you should observe the yellow highlight showing the program's progress and the values of the Fibonacci sequence appearing in the Data Segment display.
- The  icon resets the program and simulator to initial values. Memory contents are those specified within the program, and register contents are generally zero.
- The  icon is "single-step." Its complement is , "single-step backwards" (undoes each operation).

9. Observe the output of the program in the Run I/O display window:

The Fibonacci numbers are:



1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181


-- program is finished running --

10. Modify the contents of memory. (Modifying a register value is exactly the same.)

- Set a breakpoint at the first instruction of the subroutine which prints results. Use the checkbox at the left of the instruction whose address is $0x00400060 = 4194400_{ten}$.

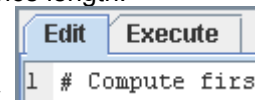
<input checked="" type="checkbox"/>	0x00400060	0x00044020	add \$8,\$0,\$4	50: print:s
-------------------------------------	------------	------------	-----------------	-------------

- Reset  and re-run  the program, which stops at the breakpoint.
- Double-click in one of the memory locations containing the computed Fibonacci numbers. The cell will be highlighted and will accept keyboard entry, similar to a spreadsheet. Enter some noticeably different value, and use the Enter key or click outside the cell to indicate that the change is complete. *Example: Memory address $0x10010020 = 268501024_{ten}$ presently contains data $0x00000022 = 34_{ten}$.*


- Click  to continue from the breakpoint. The program output includes your entered value instead of the computed Fibonacci number.




11. Open the Help  for information on MIPS instructions, pseudoinstructions, directives, and syscalls.

12. Modify the program so that it prompts the user for the Fibonacci sequence length.



- Select the Edit tab in the upper right to return to the program editor.
- The MIPS comment symbol is #. All characters on the line after the character # are ignored.
- Un-comment lines 12-19. The newly exposed program fragment will prompt the user for the length of the Fibonacci sequence to generate, in the range $2 \leq x \leq 19$. (The length of the sequence must be limited to the size of the declared space for result storage.)
- Determine the correct **syscall** parameter to perform “read integer” from the user, and insert the

parameter at line The correct **syscall** parameter may be found at Help  ... Syscall tab...read integer service. The completed line will have the form **li \$v0, 42** (where in this case 42 is not the right answer).

- Reset  and re-run  the program. The program will stop at the breakpoint you inserted previously. Continue and finish with .

8.5.5.4 MIPS Assembler Direktiven

- `.align` Align next data item on specified byte boundary (0=byte, 1=half, 2=word, 3=double)
- `.ascii` Store the string in the Data segment but do not add null terminator
- `.asciiz` Store the string in the Data segment and add null terminator
- `.byte` Store the listed value(s) as 8 bit bytes
- `.data` Subsequent items stored in Data segment at next available address
- `.double` Store the listed value(s) as double precision floating point
- `.end_macro` End macro definition. See `.macro`
- `.eqv` Substitute second operand for first. First operand is symbol, second operand is expression
- `.extern` Declare the listed label and byte length to be a global data field
- `.float` Store the listed value(s) as single precision floating point
- `.globl` Declare the listed label(s) as global to enable referencing from other files
- `.half` Store the listed value(s) as 16 bit halfwords on halfword boundary
- `.include` Insert the contents of the specified file. Put filename in quotes.
- `.kdata` Subsequent items stored in Kernel Data segment at next available address
- `.ktext` Subsequent items (instructions) stored in Kernel Text segment at next available address
- `.macro` Begin macro definition. See `.end_macro`
- `.set` Set assembler variables. Currently ignored but included for SPIM compatability
- `.space` Reserve the next specified number of bytes in Data segment
- `.text` Subsequent items (instructions) stored in Text segment at next available address
- `.word` Store the listed value(s) as 32 bit words on word boundary

8.5.5.5 MARS Interactive Debugging Features

MARS provides many features for interactive debugging through its Execute pane. Features include:

- In **Step** mode, the next instruction to be simulated is highlighted and memory content displays are updated at each step.
- Select the **Go** option if you want to simulate continually. It can also be used to continue simulation from a paused (step, breakpoint, pause) state.
- **Breakpoints** are easily set and reset using the check boxes next to each instruction displayed in the Text Segment window. *New in Release 3.8:* You can temporarily suspend breakpoints using Toggle Breakpoints in the Run menu or by clicking the "Bkpt" column header in the Text Segment window. Repeat, to re-activate.
- When running in the Go mode, you can select the simulation speed using the **Run Speed slider**. Available speeds range from .05 instructions per second (20 seconds between steps) up to 30 instructions per second, then above this offers an "unlimited" speed. When using "unlimited" speed, code highlighting and memory display updating are turned off while simulating (but it executes really fast!). When a breakpoint is reached, highlighting and updating occur. Run speed can be adjusted while the program is running.
- When running in the Go mode, you can pause or stop simulation at any time using the **Pause** or **Stop** features. The former will pause execution and update the display, as if you were stepping or at a breakpoint. The latter will terminate execution and display final memory and register values. If running at "unlimited" speed, the system may not respond immediately but it will respond.
- You have the ability to interactively **step "backward"** through program execution one instruction at a time to "undo" execution steps. It will buffer up to 2000 of the most recent execution steps (this limit is stored in a properties file and can be changed). It will undo changes made to MIPS memory, registers or condition flags, but not console or file I/O. This should be a great debugging aid. It is available anytime execution is paused and at termination (even if terminated due to exception).
- When program execution is paused or terminated, select **Reset** to reset all memory cells and registers to their initial post-assembly values. In fact, Reset is implemented by re-assembling the program.
- Memory addresses and values, and register values, can be viewed in either decimal or hexadecimal format. All data are stored in little-endian byte order (each word consists of byte 3 followed by byte 2 then 1 then 0). Note that each word can hold 4 characters of a string and those 4 characters will appear in the reverse order from that of the string literal.

- Data segment contents are displayed 512 bytes at a time (with scrolling) starting with the data segment base address (0x10010000). Navigation buttons are provided to change the display to the next section of memory, the previous, or back to the initial (home) range. A combo box is also provided to view memory contents in the vicinity of the stack pointer (contents of MIPS \$sp register), global pointer (contents of MIPS \$gp register), the heap base address (0x10040000), .extern globals (0x10000000), the kernel data segment (0x90000000), or memory-mapped IO (MMIO, 0xFFFF0000).
- Contents of any data segment memory word and almost any MIPS register can be modified by editing its displayed table cell. Double-click on a cell to edit it and press the Enter key when finished typing the new value. If you enter an invalid 32-bit integer, the word INVALID appears in the cell and memory/register contents are not affected. Values can be entered in either decimal or hexadecimal (leading "0x"). Negative hexadecimal values can be entered in either two's complement or signed format. Note that three of the integer registers (zero, program counter, return address) cannot be edited.
- Contents of cells representing floating point registers can be edited as described above and will accept valid hexadecimal or decimal floating point values. Since each double-precision register overlays two single-precision registers, any changes to a double-precision register will affect one or both of the displayed contents of its corresponding single-precision registers. Changes to a single-precision register will affect the display of its corresponding double-precision register. Values entered in hexadecimal need to conform to IEEE-754 format. Values entered in decimal are entered using decimal points and E-notation (e.g. 12.5e3 is 12.5 times 10 cubed).
- Cell contents can be edited during program execution and once accepted will apply starting with the next instruction to be executed.
- Clicking on a Labels window entry will cause the location associated with that label to be centered and highlighted in the Text Segment or Data Segment window as appropriate. Note the Labels window is not displayed by default but can be by selecting it from the Settings menu.

9 Entwicklungsprozesse

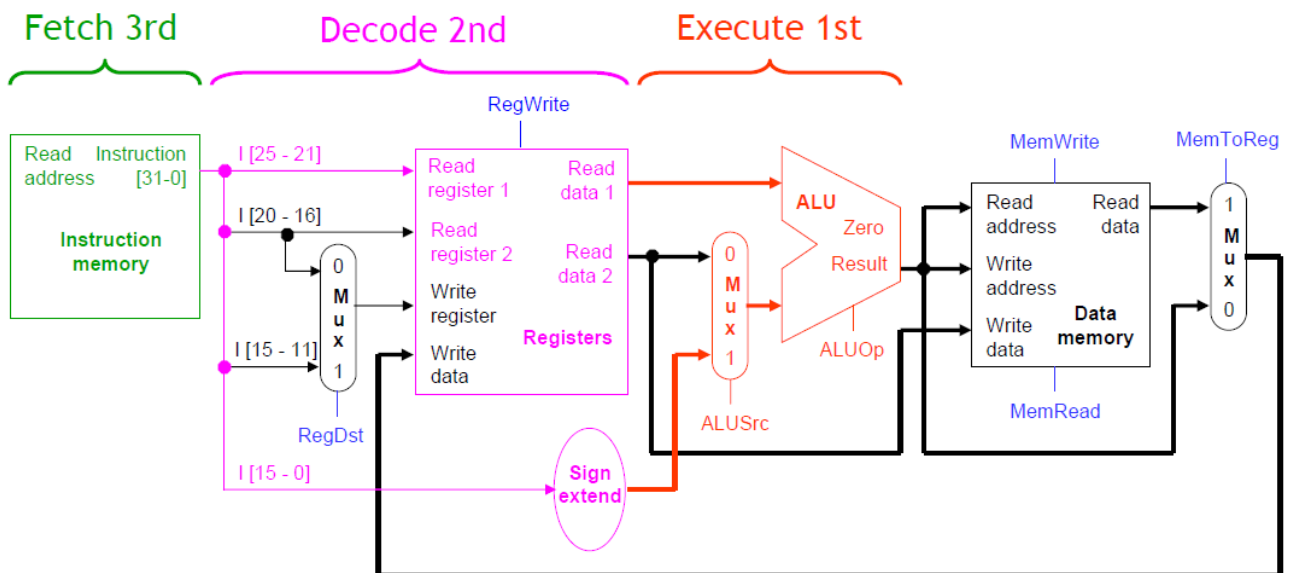
9.1 Geschwindigkeitssteigerung

Single Cycle Datenpfad ist in der Geschwindigkeit arg benachteiligt (zu lange kombinatorische Verzögerungszeiten).

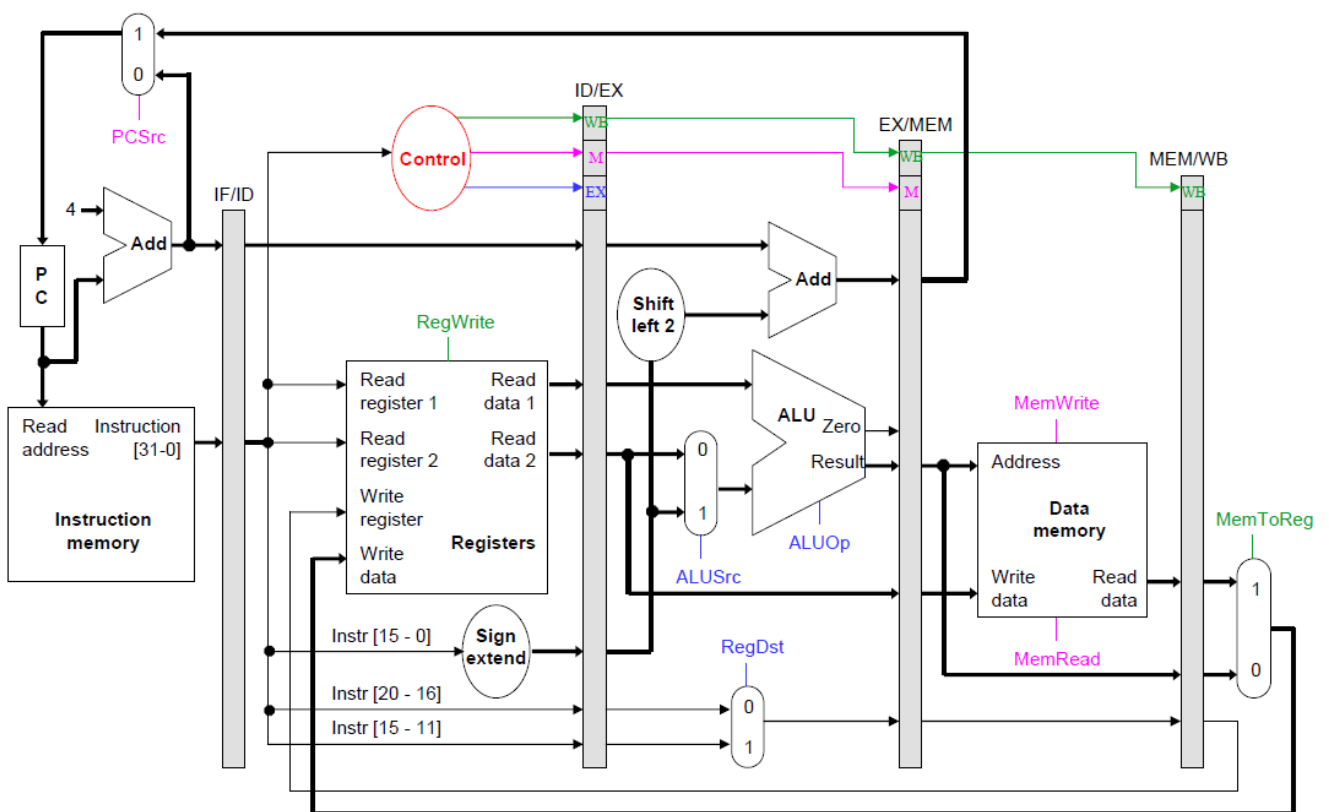
9.1.1 Phasenpipeline

Befehlsararbeitung in einer MIPS-Architektur (andere RISC-Architekturen sind vergleichbar) kann in einzelne Phasen unterteilt werden

STEP	NAME	DESCRIPTION
Instruction fetch	IF	Read instruction from memory
Instruction decode	ID	Read source registers, generate control signals
Execute	EX	Compute R-result or branch target
Memory	MEM	Read or write data to/from memory
Write back	WB	Store result in register



Daten und Steuersignale werden an den Schnittstellen der Phasen in Registern gespeichert.



9.1.1.1 Pipeline-Hazards

Three kinds of hazards conspire to make pipelining difficult.

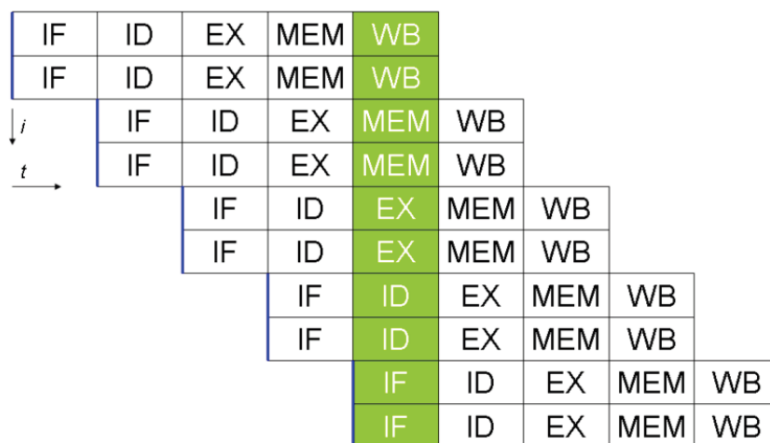
- Structural hazards result from not having enough hardware available to execute multiple instructions simultaneously.
 - These are avoided by adding more functional units (e.g., more adders or memories) or by redesigning the pipeline stages.
- Data hazards can occur when instructions need to access registers that haven't been updated yet.
 - Hazards from R-type instructions can be avoided with forwarding.
 - Loads can result in a "true" hazard, which must stall the pipeline.
- Control hazards arise when the CPU cannot determine which instruction to fetch next.
 - We can minimize delays by doing branch tests earlier in the pipeline.
 - We can also take a chance and predict the branch direction, to make the most of a bad situation.

9.1.2 Superskalarität

Weitere Geschwindigkeitssteigerung:

- Deeper Pipelines
- Dynamic Branch Prediction
- Branch Target Buffers (removing the taken branch penalty)
- Multiple Issue / Superscalar
- Statisches Scheduling (in order)
- Dynamisches, Out-of-order Scheduling

Superskalar:



- Superskalare CPUs besitzen mehrere Recheneinheiten: 4. . . 10
- In jedem Takt werden (dynamisch) mehrere Instruktionen eines konventionell linearen Instruktionsstroms abgearbeitet: $CPI < 1$
- Hardware verteilt initiierte Instruktionen auf Recheneinheiten
- Pro Takt kann *mehr als eine* Instruktion initiiert werden
- Die Anzahl wird dynamisch von der Hardware bestimmt: „*Instruction Issue Bandwidth*“
- sehr effizient, alle modernen CPUs sind superskalar
- Abhängigkeiten zwischen Instruktionen sind der Engpass, das Problem der Hazards wird verschärft (zB. RAW-Konflikt mit Forwarding nicht auflösbar)

Register Renaming:

Hardware löst Datenabhängigkeiten innerhalb der Pipeline auf; zwei Registersätze sind vorhanden:

1. Architektur-Register: „logische Register“ der ISA
2. viele Hardware-Register: „Rename Register“

dynamische Abbildung von ISA- auf Hardware-Register (register renaming)

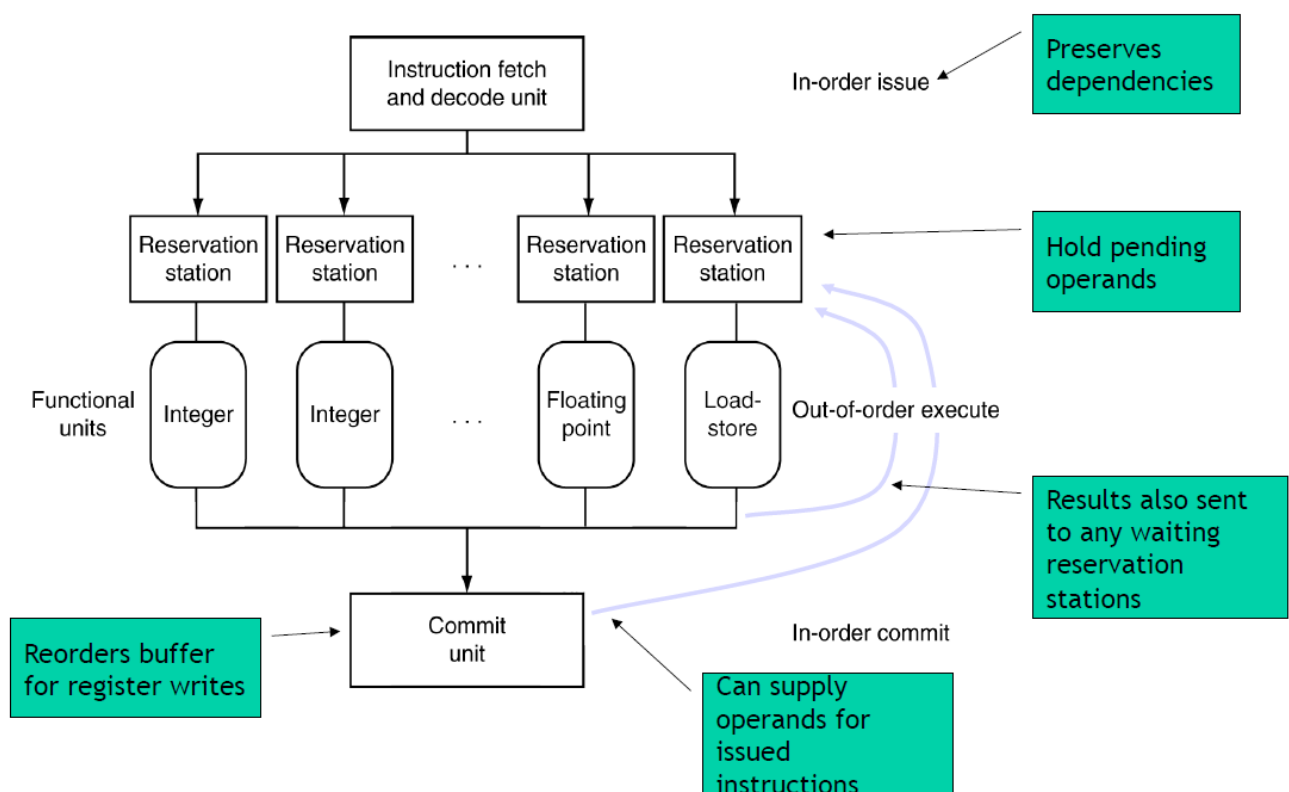
zB Originalcode (4 Cy.)	nach Renaming
tmp = a + b;	tmp1 = a + b;
res1 = c + tmp;	res1 = c + tmp1;
tmp = d + e;	tmp2 = d + e;
res2 = tmp - f;	res2 = tmp2 - f;
	tmp = tmp2;

Parallelisierung des modifizierten Codes (2 Cy)

tmp1 = a + b;	tmp2 = d + e;	
res1 = c + tmp1;	res2 = tmp2 - f;	tmp = tmp2;

Implementing Out-of-order Execution:

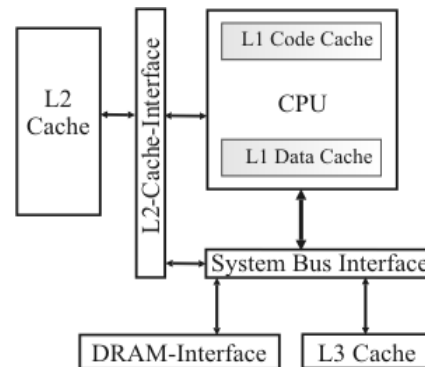
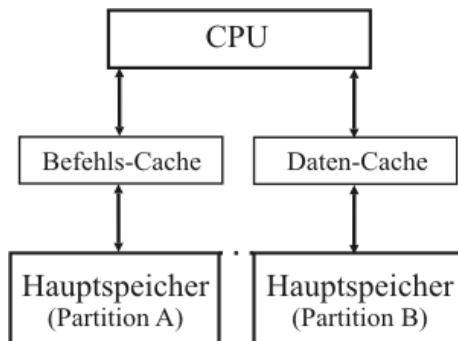
1. Fetch instructions in program order ($\leq 4/\text{clock}$)
2. Predict branches as taken/not-taken
3. To avoid hazards on registers, rename registers using a set of internal registers (>80 registers)
4. Collection of renamed instructions might execute in a window (>60 instructions)
5. Execute instructions with ready operands in 1 of multiple functional units (ALUs, FPU, Ld/St)
6. Buffer results of executed instructions until predicted branches are resolved in reorder buffer
7. If predicted branch correctly, commit results in program order (retire stage)
8. If predicted branch incorrectly, discard all dependent results and start with correct PC



9.2 Speicherorganisation

9.2.1 Cache

9.2.1.1 Cache - warum?



9.2.1.2 Direktabbildender Cache

siehe Mitschrift

9.2.1.3 Vollassoziativer Cache

siehe Mitschrift

9.2.1.4 Mehrwege-assoziativer Cache

siehe Mitschrift

9.2.1.5 Cache Ersetzungsstrategien

LRU Least Recently Used: der Eintrag, auf den am längsten nicht zugegriffen wurde, wird ersetzt. Dazu braucht man einen Zähler zu jedem Eintrag - aufwendige Verwaltung

Random: (pseudo) zufällige Ersetzung eines Eintrags; Cache Hit Rate ist nicht wesentlich schlechter als bei LRU.

Was macht man, wenn Cache Line von der CPU modifiziert wird?

Write Through: Hauptspeicher wird sofort aktualisiert - eventuell unvorteilhaft, wenn nacheinander mehrere Einträge in einer Line modifiziert werden.

Write Back: Line wird erst dann in den HS zurückgeschrieben, wenn die Line gebraucht wird; dazu muss für jede Line ein „dirty - Bit“ verwaltet werden.

9.2.1.6 Cache in Mehrprozessorsystemen

Problem: mehrere CPUs mit je einem Cache und gemeinsamer Hauptspeicher. Wie wird sichergestellt, dass jede CPU aus ihrem Cache nur gültige Einträge liest?

→ Cache Kohärenzprotokoll ist notwendig.

→ Snoop - Logik, damit andere CPUs „mitbekommen“, was mit Cache- bzw HS-Inhalt passiert ist.

9.2.1.7 Cache Kohärenz Protokolle

Einfachstes Kohärenz-Protokoll: **Write Through**

alle Schreibzugriffe werden auch gleich in den Hauptspeicher durchgeschrieben.

Vorteil: einfach; Nachteil: Busverkehr bei jedem Schreibvorgang

Aktion	Cache	Bus	andere Caches
Read Hit	Read	----	----
Read Miss	Read	Line Fill	----
Write Hit	Write	Write	Snoop / Invalidate
Write Miss	-----	Write	Snoop / Invalidate

MESI- Protokoll

4 Zustände sind für jede Cacheline möglich

Invalid: Eintrag ungültig, Lesen führt zu Cache Miss und Allocate (Laden), Schreiben führt zu Cache Miss mit Write Through (→ weiterhin invalid)

Exclusive (Clean): Eintrag befindet sich nur in dem **einen** Cache und im HS: Lesen: read hit; Schreiben: write hit (→ modified)

Shared: Eintrag befindet sich in dem einen Cache und möglicherweise auch noch in anderen Caches: Lesen: read hit, schreiben: write hit , write through (→ .exclusive)

Modified (dirty): Ein dem einen Cache befindet sich nach einem Schreibvorgang der einzig gültige Eintrag; HS ist nicht gültig. Wenn andere CPUs den Eintrag brauchen, müssen sie ihn über Snooping anfordern, damit er in den HS zurückgeschrieben wird (write back) und in den Zustand Shared oder Exclusive kommt.

Cache line state	M (modified)	E (exclusive)	S (shared)	I (invalid)
This cache line is valid?	Yes	Yes	Yes	No
The memory copy is..	..out of date	.. valid	.. valid	----
Copies exist in caches of other Prozessors	No	No	Maybe	Maybe
A write to this line does not go to the bus	.. does not go to the bus	.. goes to the bus and updates cache	.. goes directly to the bus

Action	Current state	Next state	Bus Activity
Read	M	M	None (Read Hit)
Read	E	E	None (Read Hit)
Read	S	S	None (Read Hit)
Read	I	S / E	Line fill (Read Miss) + send Inq (intend to read)
Write	M	M	None (Write Hit)
Write	E	M	None (Write Hit)
Write	S	S / E	Write Through(Write Hit) + send INV to other Caches
Write	I	I	Write Through(Write Hit) + send INV + wait for line to be written by other

Was macht Snooping? Snoop-Signale werden aus dem Steuerbussen und den Adressbussen der Caches abgeleitet; Busmaster kann auch noch zusätzlich „invalidate“ Signal erzeugen.

Action	Current State	Next State	Bus Activity
Snoop Read	M	S	Write Back + send Hit & HitM
Snoop Read	E	S	Send Hit
Snoop Read	S	S	Send Hit
Snoop Read	I	I	do nothing
Snoop Write	M	I	Write Back + send Hit / HitM + invalidate owned copy
Snoop Write	E	I	Invalidate owned copy
Snoop Write	S	I	Invalidate owned copy
Snoop Write	I	I	Invalidate owned copy

Weiterentwicklung für Multiprozessorsysteme: **MOESI Protokoll**

O= owned = modified shared.

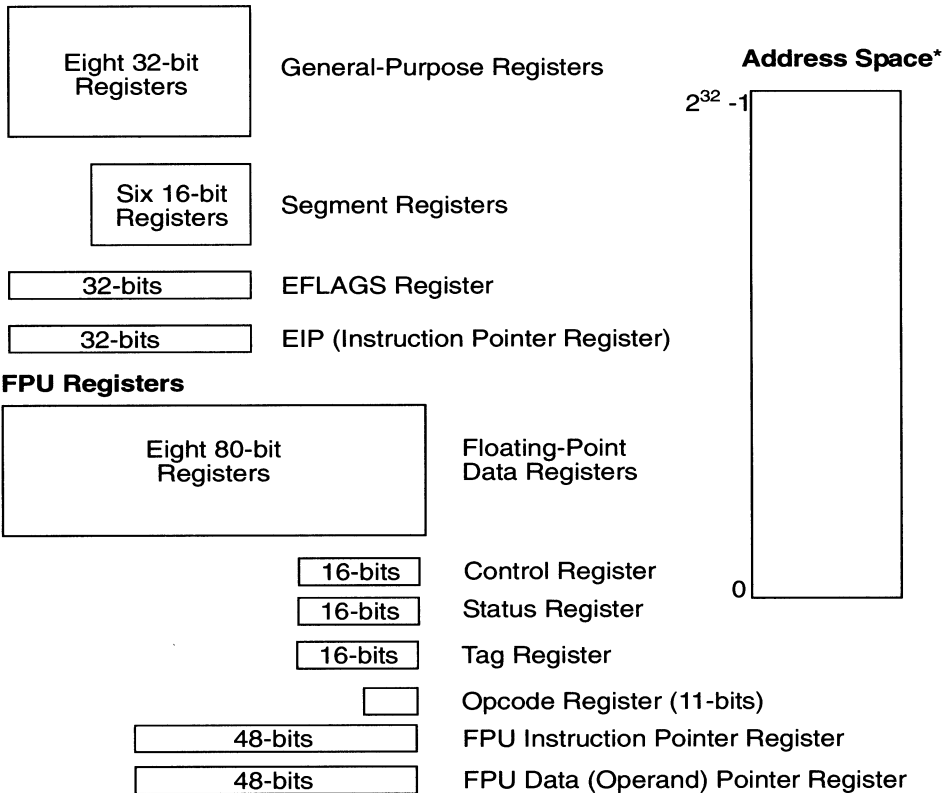
Direktes Kopieren von modifizierten Cachelines von Cache zu Cache, ohne dass in den Hauptspeicher geschrieben wird erhöht die Performance.

9.2.2 Virtueller Speicher

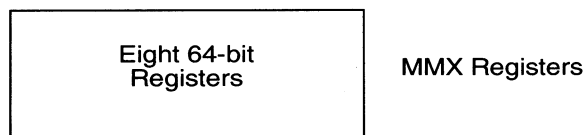
Bei Multitasking arbeitet jeder Prozess auf eigenem logischen Code- und Datenadressraum (Sicherheit!). Alle logischen Adressräume bilden den virtuellen Speicher.

9.2.2.1 IA32 Programmiermodell

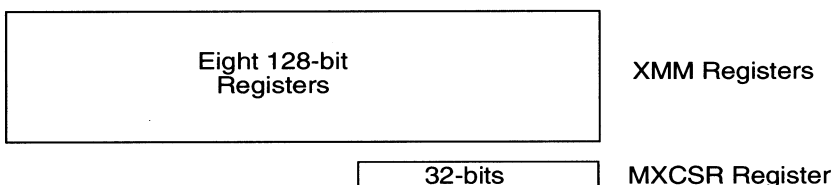
Basic Program Execution Registers



MMX Registers



SSE and SSE2 Registers



Zusätzliche Register:

- I/O Ports
- Control registers CR0 .. CR4
- Memory Management Registers GDTR, IDTR, LDTR,
- Debug Registers DR0 .. DR7
- Memory type range registers MTRRs
- Machine specific registers MSRs
- Performance monitoring counters

General-Purpose Registers

31	16	15	8	7	0	16-bit	32-bit
			AH	AL		AX	EAX
			BH	BL		BX	EBX
			CH	CL		CX	ECX
			DH	DL		DX	EDX
			BP				EBP
			SI				ESI
			DI				EDI
			SP				ESP

64-Bit Mode von IA32:

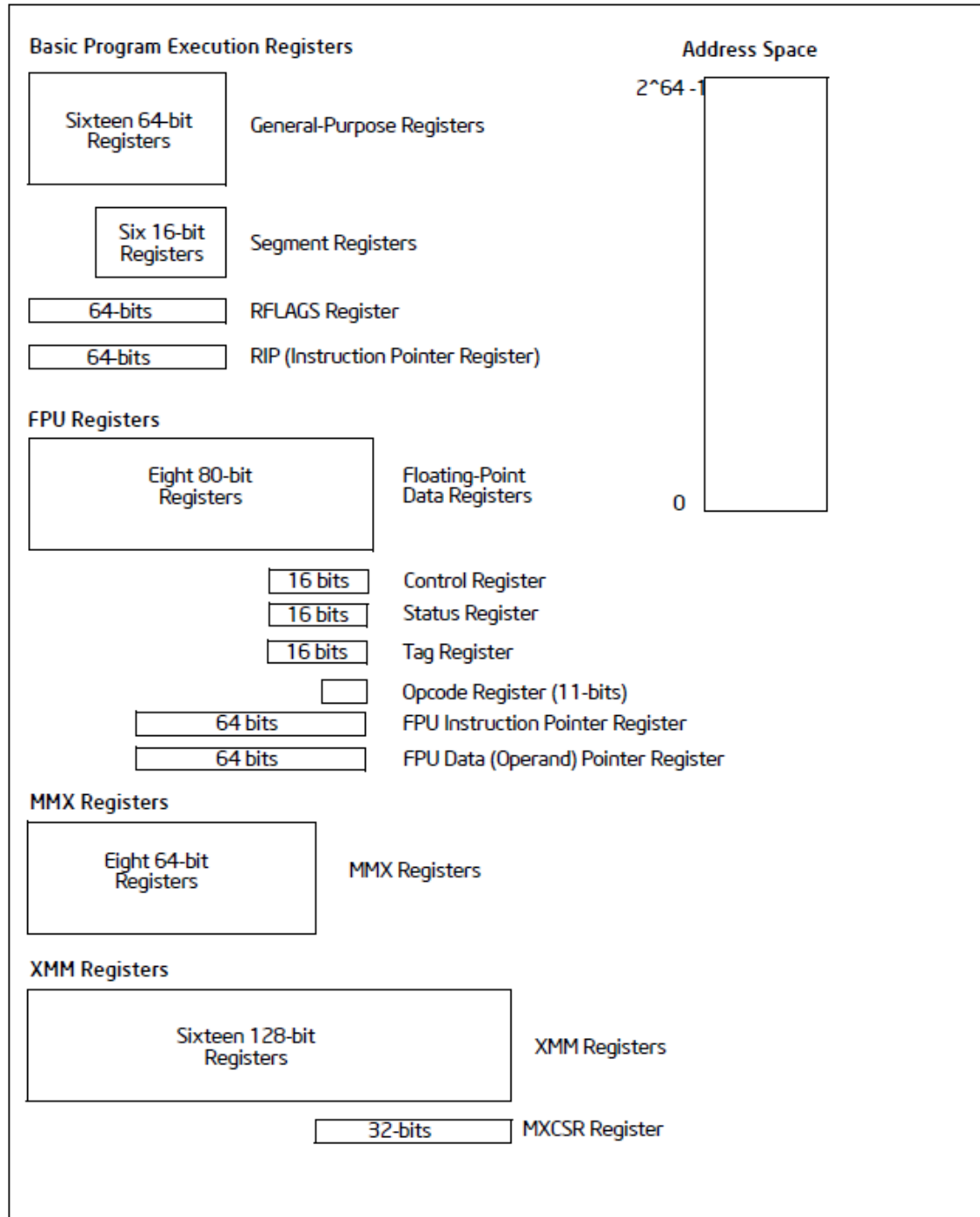


Figure 3-2. 64-Bit Mode Execution Environment

9.2.2.2 Verwaltung des virtuellen Speichers:

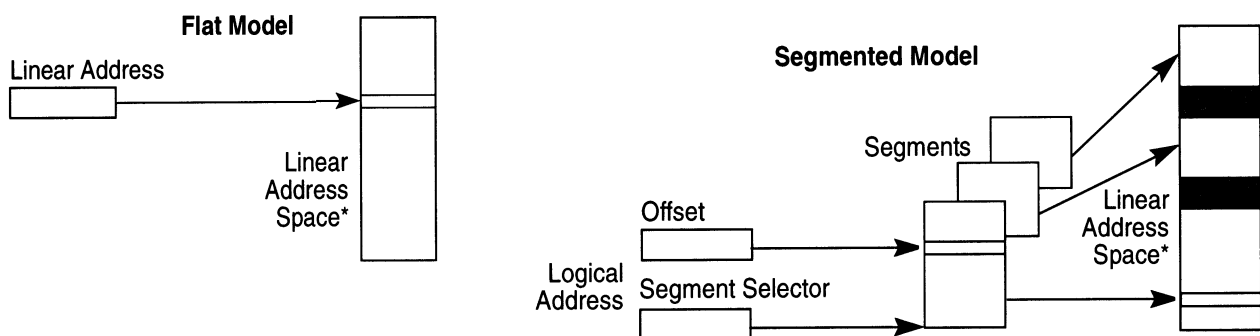
Segmente für Code, Daten, Stack etc. jeweils für einen Prozess

Swapping: Ein- und Auslagern kompletter Segmente, wenn physikalischer Speicher knapp wird (Vorgang ist zeitaufwändig) daher:

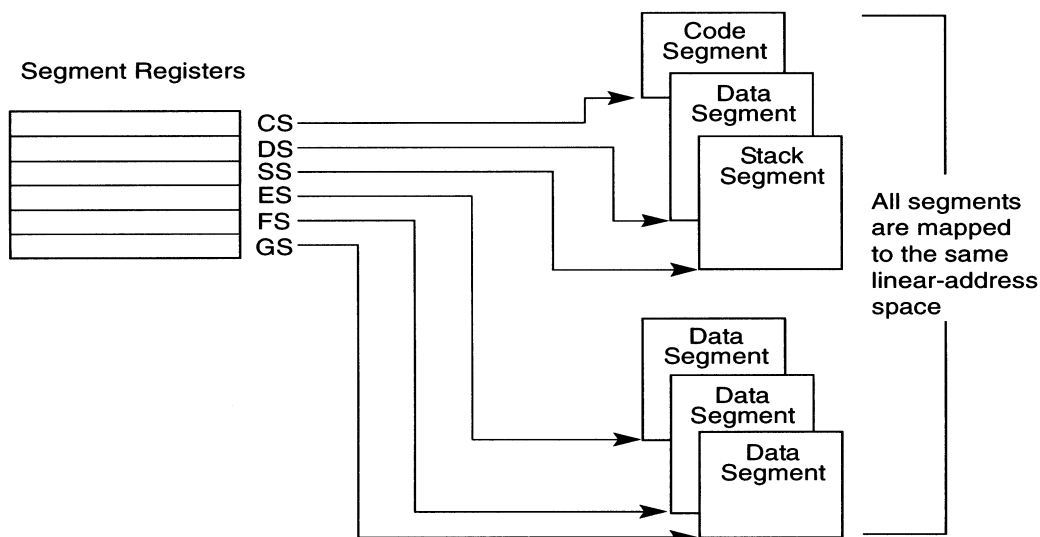
Paging: Unterteilung der Segmente in lauter gleichgroße "Kacheln", die leicht ein- und ausgelagert werden können. Nicht mehr ganze Segmente werden bewegt, sondern nur die in den Segmenten gerade benötigten Pages.

9.2.2.3 IA32 Adressübersetzung

real mode (gibt es praktisch nicht mehr): Segment:Offset (je 16 Bit) bilden gemeinsam eine 20 Bit Adresse (Segment um 4 Bit nach links verschoben + Offset).

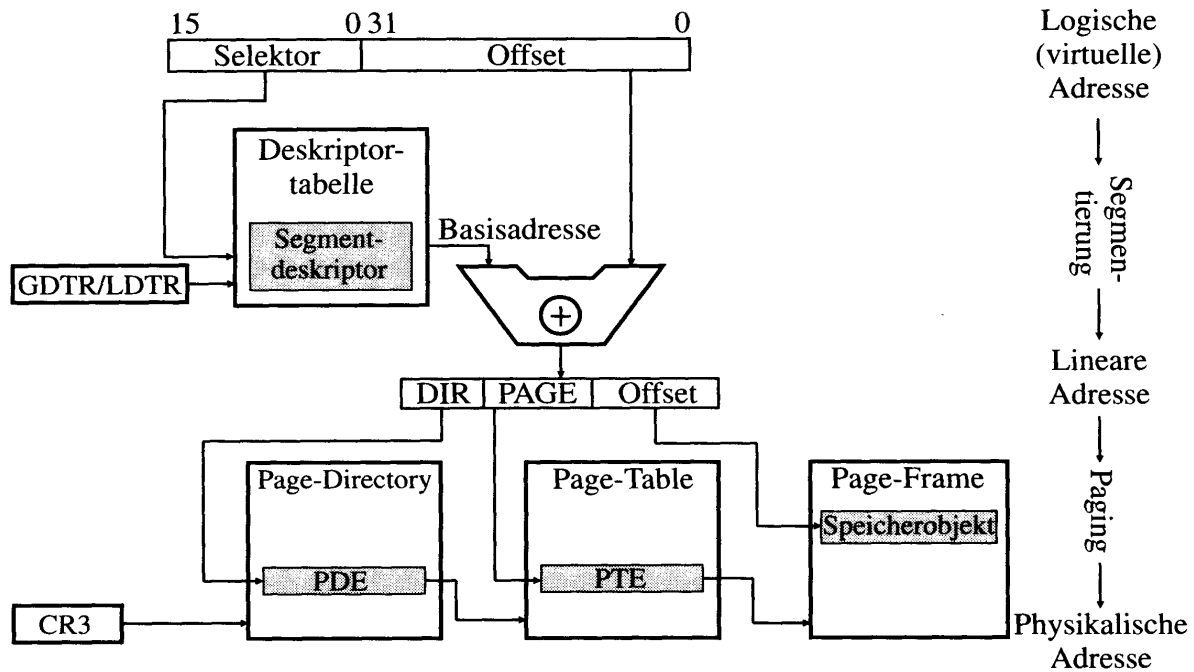


Adressbildung im real mode:

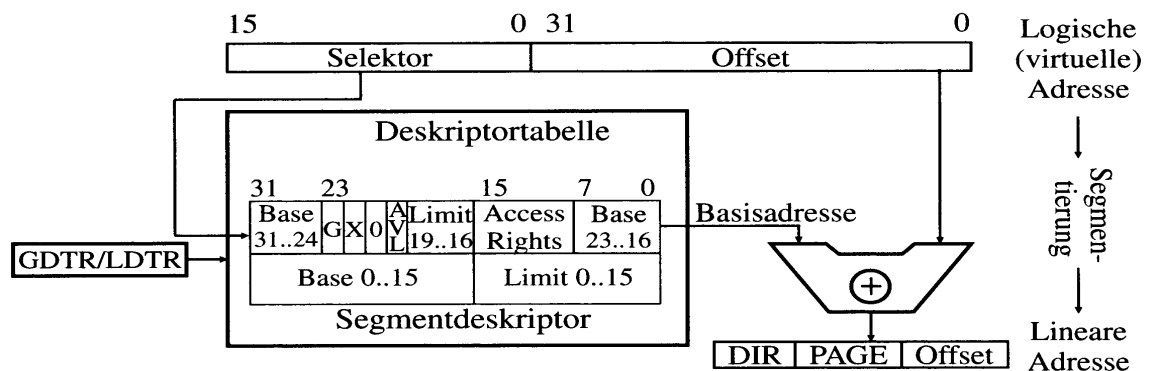


protected Mode: segmentierter Speicher, aber der Segment-Selector aus dem Segmentregister wird völlig anders verwendet; er zeigt nicht mehr direkt in das Segment sondern in die Segment-Deskriptoren Tabelle jedes Programms.

- Logische Adresse (=virtuelle Adresse) = 48 Bit breit: 16 Bit SELECTOR, 32 Bit OFFSET
- SELECTOR besteht aus 13 Bit INDEX und GDT/LDT -Bit (GDT/LDT: global, local descriptor table) und 2 Bit RPL (requested privileg level)
- GDTR bzw LDTR Prozessor-Register liefern Start der Deskriptoren-Tabelle mit lauter 8 Byte langen Einträgen (=Segmentdeskriptoren)

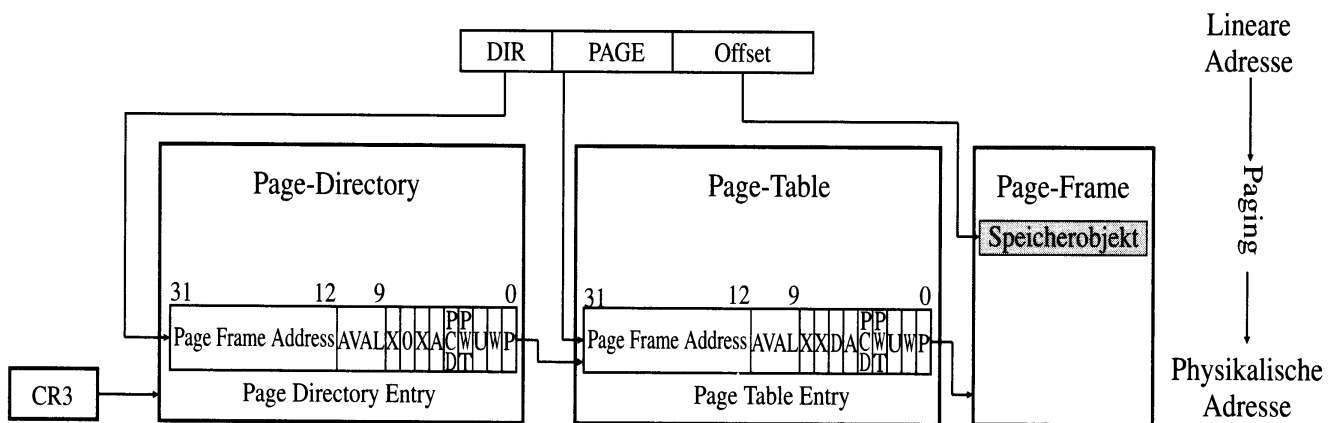


Segmentdeskriptor - Basisadresse (32 Bit) + OFFSET aus logischer (virtueller) Adresse liefert die lineare Adresse; zusätzlich stehen noch weitere Infos im Segmentdeskriptor: Access rights, Limit...



wenn paging abgeschaltet ist, dann ist lineare Adresse = physikalische Adresse -> fertig. - sonst: lineare Adresse wird interpretiert als: DIR (10 Bit) : PAGE (10 Bit) : OFFSET (12 Bit)

DIR zeigt ins Page Directory (Start von Prozessorregister CR3 adressiert)

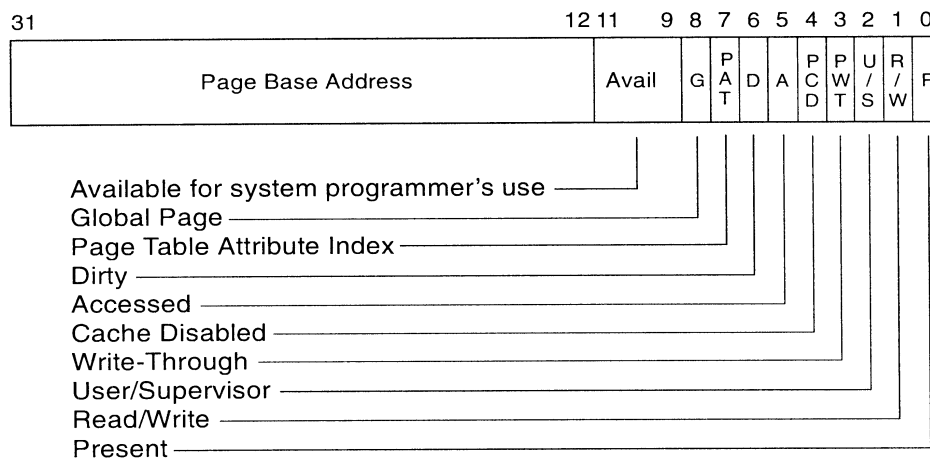


Jeder PDE (Page Directory Entry) zeigt auf eine Page Table Base Address (+ 12 weitere Statusbits)
PAGE zeigt in die Page Table.

Jeder PTE (Page Table Entry) zeigt auf eine Page Base Address = Startadresse eines Pageframe.
(+12 weitere Statusbits wie vorhin)

OFFSET Adressiert schließlich das Speicherobjekt im Pageframe.

Page-Table Entry (4-KByte Page)



Ist eine so adressierte Page nicht im Speicher: Page Fault = Exception 14; im CR2 steht die virtuelle Adresse, Error Codes auf dem Stack und das Betriebssystem muss page laden.

TLB (translation lookaside buffer, on chip) speichern zu linearen Adressen die Startadresse der page
-> Zugriff wesentlich schneller möglich.

9.3 Aktuelle Beispiele für Prozessorarchitekturen

aktuelle Beispiele

10 Leistungsbewertung von Ra

10.1 Leistung

Meist wird Rechengeschwindigkeit betrachtet; andere Kriterien: Stromverbrauch, Zuverlässigkeit, ...

10.2 Ziele der Leistungsbewertung

- Auswahl eines Rechnersystems
- Veränderung der Konfiguration eines bestehenden Systems
- Optimierung des Systems
- Entwurf von Rechnern

10.3 Wann ist ein Rechner schneller?

10.3.1 Benutzer eines Arbeitsplatzrechners:

„Ein Rechner A ist schneller als ein Rechner B, wenn ein Programm auf A weniger Zeit benötigt.“

- Reduzierung der Antwortzeit (response time) oder Ausführungszeit (execution time) (= Zeit zwischen dem Beginn und dem Ende eines Ereignisses, einer Aufgabe)

"A ist n-mal so schnell wie B" bedeutet: $(1 / \text{Ausführungszeit auf A}) / (1 / \text{Ausführungszeit auf B}) = n$

10.3.2 Für Rechenzentrumsleiter:

„Ein Rechner A ist schneller als ein Rechner B, wenn A in einer Stunde mehr Aufträge (Jobs) erledigt.“

- Erhöhung des Durchsatzes (throughput) = Anzahl der ausgeführten Aufgaben in einem gegebenen Zeitintervall

"Durchsatz von A ist m-mal so hoch wie der von B": Die Anzahl der erledigten Aufgaben auf A ist m-mal die Anzahl der erledigten Aufgaben auf B in einer Zeiteinheit.

10.4 Ausführungszeit (execution time)

10.4.1 Wall-clock time, response time, elapsed time

Latenzzeit für die Ausführung einer Aufgabe; schließt den Speicher- und Plattenzugriff, Ein-/ Ausgabe etc. mit ein.

10.4.2 CPU Time

Zeit, in der die CPU arbeitet

- User CPU Time: Zeit, in der die CPU ein Programm ausführt
- System CPU Time: Zeit, in der die CPU Betriebssystemaufgaben ausführt, die von einem Programm angefordert werden

Beispiel Unix Time Kommando:

90.7s 12.9s 2:39 65%

System CPU time

User CPU time |

Elapsed time |

%CPU time an der Elapsed time: $(90.7s + 12.9s) / 159s =$

0.65

10.5 Programmunabhängige Leistungsmodelle

10.5.1 Taktfrequenz

ungeeignete Metrik, wird in Werbung verwendet; zB: Leistungsanforderung "CPU mit mindestens 1.5 GHz"

10.5.2 MIPS

Million Instructions per second

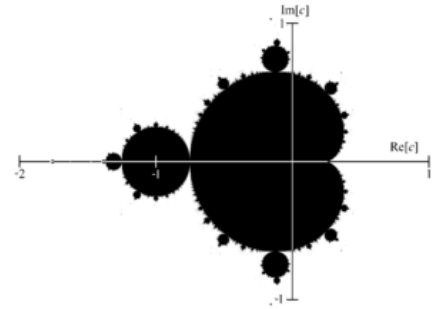
$$\text{DEF: MIPS} = N(\text{instr}) / 1\text{E6 pro s}$$

Bsp 1 (Mandelbrotmenge= Apfelmännchen)

Die Mandelbrot-Menge ist die Menge aller [komplexen Zahlen](#) c , für welche die [rekursiv definierte Folge](#) komplexer Zahlen z_0, z_1, z_2, \dots mit dem Bildungsgesetz

$$z_{n+1} := z_n^2 + c \text{ und der Anfangsbedingung } z_0 := 0$$

[beschränkt](#) bleibt, das heißt, der [Betrag](#) der Folgenglieder wächst nicht über alle Grenzen (Wikipedia)



CPI (Cycles per Instruction), **IPC** (Instructions per Cycle)

$$\text{MIPS} = \text{IPC} * \text{fcy} / 1\text{E6} = \text{fcy} / 1\text{E6} * 1/\text{CPI} = 1 / (1\text{E6} * \text{tcy} * \text{CPI})$$

Bsp 2 (Vergleich P100, P4)

Arten von MIPS:

- native MIPS (tatsächliche CPI entsprechend Befehlsmix)
- peak MIPS (CPI min)
- relative MIPS (bezogen auf Referenzrechner) VAX 11/780 = IBM 370 158-3 = 1

wenn man Operationen genauer spezifizieren will: **FLOPS** (floating point..); **OPS** (operations ...)

MIPS sind als Leistungsindikator wenig gut geeignet ("*Meaningless Indicator of Processor Speed*")

10.6 Programmabhängige Leistungsmodelle

Leistung = Performance: gemessen in Anzahl der Programmausführungen pro Zeiteinheit

10.6.1 Performance

$$P(\text{RA}) = 1 / T(\text{ProgX}, \text{RA})$$

$T(\text{ProgX}, \text{RA})$: Ausführungszeit für Programm Prog X auf Rechner A

Ausführungszeit

$$T(\text{ProgX}, \text{RA}) = N_{\text{inst}}(\text{ProgX}) * \text{CPI}(\text{ProgX}) * \text{tcy}(\text{RA})$$

Optimierung von T

- N_{inst} zB IA32 CISC-Befehle; wieviele μOPs pro CISC-Befehl? (ca. 1.3)
- $\text{CPI}(\text{ProgX})$
 - ohne Superskalarität: theoretisch 1.0 erreichbar (1.2- 1.5)
 - n-Fach Issue-Architekturen (zB 3-fach: PIII, P4, Athlon): $\text{CPI}_{\text{min}} = 1/3$ (0.6 ... 0.8)
 - Benchmarks (SpecInt, SpecFP) so, dass Code und Daten nicht in Cache passen
- $\text{tcy}(\text{RA})$ altbekannt
- aber häufige Grenze für tcy : notwendige Speicherbandbreite:
 $\text{SB}_{\text{max}} = \text{MIPS} * \text{Sreq}(\text{RA})$

SB: Speicherbandbreite
 Sreq: Byte/Befehl (zB lt. alter Studie: 3.3Bef + 5.3 Daten pro Befehl, viel load/store))
 IA32, Power mit wenig load/store: 5.5 Byte/Befehl

Bsp 3**10.6.2 relative Performance**

Vergleich zweier Rechenanlagen

$$P(RA) / P(RB) = 1 / T(\text{ProgX}, RA) / 1 / T(\text{ProgX}, RB)$$

$$\text{mit } T(\text{ProgX}, RB) = N_{\text{takte}}(\text{ProgX}, RB) * t_{cy}(RB) = N_{\text{inst}} * CPI * T_{cy}$$

Bsp4 (A,B) ; Bsp5 (A,B) ; Bsp 6 (Befehlsklassen, CPI mittel berechnen)

10.6.3 Teiloptimierung

$$T_{\text{neu}}(\text{ProgX}) = T(\text{ProgX}, \text{optimierbar}) / VF + T(\text{ProgX}, \text{Rest})$$

= Amdahlsches Gesetz

VF = Verbesserungsfaktor

Bsp 7**10.7 Parallelverarbeitung**

andere Form des Amdahlschen Gesetzes für p parallele Rechner

$$T_{\text{neu}}(\text{ProgX}, p) = T(\text{ProgX}, \text{parallelisierbar}) / p + T(\text{ProgX}, \text{nicht parallelisierbar}) + OH(p)$$

OH: (Kommunikations-) Overhead

10.7.1 Speedup SU

Verbesserung durch Parallelisierung:

$$SU(p) = T(\text{ProgX}, \text{alt}) / T(\text{ProgX}, \text{neu}) = P(\text{ProgX}, \text{neu}) / P(\text{ProgX}, \text{alt})$$

Effizienz E von p parallelen Prozessoren

$$E(p) = \text{Speedup}(p) / p$$

Bsp 8 (p Prozessoren)

Speedup kann (sehr selten) auch größer als p werden (Cache Effekte bei Speicherhierarchie)

10.8 Performancesteigerung durch Architekturänderungen

Bsp 9: 3 Architekturvorschläge A,B,C; Referenzprogramm hat Teile PT1 bis PT5

Programmteil	Arch A	Arch B	Arch C	Anteil an Rechenzeit
PT1	2x so schnell	30% langsamer	4x so schnell	20 %
PT2	1,2x langsamer	4x so schnell	20% langsamer	18 %
PT3	gleich	2x so schnell	30% schneller	20 %
PT4	20% schneller	gleich	25% langsamer	20 %
PT5	2,3x langsamer	3x so schnell	1,2x langsamer	Rest= 22 %

ges: Speedup, theoretischer Speedup bei Kombination der schnellsten Möglichkeiten (Architektur "D")

Bsp 10: 2 Codevorschläge, $t_{cy} = 1 / 500 \text{ MHz}$

ges: CPI mittel, native MIPS, rel. Performance

Codevorschlag	Befehlsklasse A, CPI=1	Befehlsklasse B, CPI=2	Befehlsklasse C, CPI=3
X	5 E 9	1 E 9	1 E 9
Y	10 E 9	1 E 9	1 E 9

Bsp 11: ISA-Performance

Zwei Implementierungen M1: fcy=50 MHz, M2: fcy=75 MHz; ges: Peak MIPS; MIPS mittel

Instruktions-Klassen	CPI (M1)	CPI (M2)	Häufigkeit
A	1.4	2	40 %
B	2	1.7	43 %
C	5.2	4	11 %
D	1	8	6 %

10.9 Kosten

Kenngröße: Preis - Leistungs-Verhältnis (price-performance ratio; value for money))

$$\text{VFM} = \text{Performance} / \text{Preis}$$

Bsp 12: Vergleich zweier Systeme

System RA: Pr (RA) = 10 000 €; System RB: Pr (RB) = 15 000 €.

Anm: Pr = Preis

Programm	T (ProgX, RA)	T (ProgX, RB)
Prog1	11 s	5 s
Prog2	4 s	4 s

ges: Günstigstes System, wenn Prog1 und Prog2 jeweils 1 mal ausgeführt werden.

Bsp 13: wie Bsp12, nur Prog1 betrachtet;

- RA wird verbessert zu RAneu: Integer doppelt so schnell, Load/Store 60% schneller.
- Verteilung im Prog1: 30% Integer, 47% Load/Store, 23% Rest

ges: Vergleich RAneu und RB

10.10 Benchmarks

Def: **Benchmark** = strictly defined set of operations, returns some sort of result (a metric).

10.10.1 Klassen von Benchmarks:

- Real program Benchmarks
- CPU oder Kernel (zB Linpack)
- Synthetic Benchmark (zB Whetstone, Dhrystone)
- I/O benchmarks
- Parallel benchmarks
- System Benchmarks (zB SysMark, SPECWeb..)

10.10.2 Whetstone

MWIPS (million whetstone instructions per second); 1972 ALGOL60 Programme;

Inhalt: hauptsächlich standard FP Operationen (sin, alog, abs...)

Werte zB: P4/3GHz: ca. 4000 (9.2s); P3/650: 467 (78s); NEC SX6i: 14000 (4.3s)

10.10.3 Dhrystone

(1984) <http://www.roylongbottom.org.uk/>

DMIPS : bezogen auf 100% = VAX 11//85 = 1857 Dhrystone/s ca. 1MIPS

Inhalt: synthetisch; Integer- + String- (nur etwa 100) Anweisungen; leicht manipulierbar durch guten Compiler.

10.10.4 Linpac

Grundlage für TOP500 Liste (www.top500.org): Lineares Gleichungssystem, gauß'sche Elimination. N100 MFLOPS, N1000 MFLOPS, PeakMFLOPS; skaliert sehr gut für Parallelrechner;

1.Platz Top500 Nov2007: IBM BlueGene: Main Memory 73728 GB, Processor PowerPC 440 700 MHz (2.8 Gflops), 106 496 nodes (212 992 Procs), 478.2 teraFLOPS.

10.10.5 BAPCo

Business Application Performance Corporation ([Microsoft](http://www.babco.com), [Intel](http://www.babco.com), [AMD](http://www.babco.com), [Dell](http://www.babco.com), [Hewlett-Packard](http://www.babco.com), [NVIDIA](http://www.babco.com), [Lenovo](http://www.babco.com), [Apple](http://www.babco.com). <http://babco.com>)

Benchmark für Windows-Rechner: SysMark, WebMark, MobileMark

10.10.6 SPEC:

Standard Performance Evaluation Corporation <http://www.spec.org>

Benchmark kostet etwa 800\$ (200\$ f. Edu)

aktuell: SPEC CPU2006 besteht aus CINT2006 und CFP2006

10.10.7 CINT2006

- [400.perlbench](#) C Programming Language Derived from Perl V5.8.7. The workload includes SpamAssassin, MHonArc (an email indexer), and specdiff (SPEC's tool).
- [401.bzip2](#) C Compression Julian Seward's bzip2 version 1.0.3, modified to do most work in memory, rather than doing I/O.
- [403.gcc](#) C C Compiler Based on gcc Version 3.2, generates code for Opteron.
- [429.mcf](#) C Combinatorial Optimization Vehicle scheduling. Uses a network simplex algorithm (which is also used in commercial products) to schedule public transport.
- [445.gobmk](#) C Artificial Intelligence: Go Plays the game of Go, a simply described but deeply complex game.
- [456.hmmer](#) C Search Gene Sequence Protein sequence analysis using profile hidden Markov models
- [458.sjeng](#) C Artificial Intelligence: chess A highly-ranked chess program that also plays several chess variants.
- [462.libquantum](#) C Physics / Quantum Computing Simulates a quantum computer, running Shor's polynomial-time factorization algorithm.
- [464.h264ref](#) C Video Compression A reference implementation of H.264/AVC, encodes a videostream using 2 parameter sets. The H.264/AVC standard is expected to replace MPEG2
- [471.omnetpp](#) C++ Discrete Event Simulation Uses the OMNet++ discrete event simulator to model a large Ethernet campus network.
- [473.astar](#) C++ Path-finding Algorithms Pathfinding library for 2D maps
- [483.xalancbmk](#) C++ XML Processing A modified version of Xalan-C++, which transforms XML documents to other document types.

10.10.8 CFP2006

- [410.bwaves](#) Fortran Fluid Dynamics Computes 3D transonic transient laminar viscous flow.
- [416.gamess](#) Fortran Quantum Chemistry. Gamess implements a wide range of quantum chemical computations.
- [433.milc](#) C Physics / Quantum Chromodynamics A gauge field generating program for lattice gauge theory programs with dynamical quarks.
- [434.zeusmp](#) Fortran Physics / CFD ZEUS-MP is a computational fluid dynamics code developed at the Laboratory for Computational Astrophysics (NCSA, University of Illinois at Urbana-Champaign) for the simulation of astrophysical phenomena.
- [435.gromacs](#) C, Fortran Biochemistry / Molecular Dynamics Molecular dynamics, i.e. simulate Newtonian equations of motion for hundreds to millions of particles. The test case simulates protein Lysozyme in a solution.
- [436.cactusADM](#) C, Fortran Physics / General Relativity Solves the Einstein evolution equations using a staggered-leapfrog numerical method
- [437.leslie3d](#) Fortran Fluid Dynamics Computational Fluid Dynamics (CFD) using Large-Eddy Simulations with Linear-Eddy Model in 3D. Uses the MacCormack Predictor-Corrector time integration scheme.
- [444.namd](#) C++ Biology / Molecular Dynamics Simulates large biomolecular systems. The test case has 92,224 atoms of apolipoprotein A-I.
- [447.dealii](#) C++ Finite Element Analysis deal.II is a C++ program library targeted at adaptive finite elements and error estimation. The testcase solves a Helmholtz-type equation with non-constant coefficients.
- [450.soplex](#) C++ Linear Programming, Optimization Solves a linear program using a simplex algorithm and sparse linear algebra. Test cases include railroad planning and military airlift models.
- [453.povray](#) C++ Image Ray-tracing Image rendering. The testcase is a 1280x1024 anti-aliased image of a landscape with some abstract objects with textures using a Perlin noise function.
- [454.calculix](#) C, Fortran Structural Mechanics Finite element code for linear and nonlinear 3D structural applications. Uses the

- SPOOLES solver library.
- [459.GemsFDTD](#) Fortran Computational Electromagnetics Solves the Maxwell equations in 3D using the finite-difference time-domain (FDTD) method.
- [465.tonto](#) Fortran Quantum Chemistry An open source quantum chemistry package, using an object-oriented design in Fortran 95. The test case places a constraint on a molecular Hartree-Fock wavefunction calculation to better match experimental X-ray diffraction data.
- [470.lbm](#) C Fluid Dynamics Implements the "Lattice-Boltzmann Method" to simulate incompressible fluids in 3D
- [481.wrf](#) C, Fortran Weather Weather modeling from scales of meters to thousands of kilometers. The test case is from a 30km area over 2 days.
- [482.sphinx3](#) C Speech recognition A widely-known speech recognition system from Carnegie Mellon University

10.11 Performance-Analyse und -Simulation

10.11.1 Monitoring Funktionen und SW-Analyse Werkzeuge

(code-) profiler

10.11.2 VTune

"Der **VTune**™ Performance Analyzer analysiert Ihre Software und gibt Ihnen einen systemweiten Überblick. Zusätzlich geht er hinunter bis zu einer bestimmten Funktion, einem Modul oder einem einzelnen Befehl innerhalb Ihres Programms.

Dazu beobachtet der VTune™ Analyzer die Aktivitäten eines Programms (non-intrusive sampling) und ermittelt das entsprechende Stück Quellcode. Der Benutzer kann so auf einfache Weise bottlenecks also "Flaschenhälse" oder "Engstellen" in seinem Programm erkennen und gezielt beheben. Gerade durch die Darstellung auf Quellcodeebene, bis hinunter auf die Ebene eines einzelnen Befehls, wird ein enormer Produktivitätszuwachs erzielt. In der Praxis wird aus intelligentem Raten, wo sich denn eine Optimierung lohnen könnte, die zielgerichtete und effiziente Optimierung genau der Programmteile, wo dies einen maximalen Performancevorteil bringt." (Werbetext Intel)

10.11.3 SimpleScalar

OpenSource zur Modellierung + Simulation von Mikroprozessor Architekturen (und Mikroarchitekturen); ca. 10 Zeilen Code für Definition einer Instruktion; auch Cache, OOO simulierbar (letzte Version von 2001?).

Tutorial: http://www.simplescalar.com/docs/simple_tutorial_v4.pdf

10.11.4 RSIM

(Hughes et al. 2002; Pai et al. 1997a) is a simulator primarily targeted to study shared-memory cache coherent (cc-NUMA) multiprocessor architectures built from processors that aggressively exploit instruction-level parallelism (ILP).

Port für X86: <http://ditec.um.es/gacop/tools/rsim-x86/download.html>

Manual (gut für Überblick über Performance Simulation):

<http://ditec.um.es/gacop/tools/rsim-x86/manuals/html/index.html>

10.11.5 PCSIM

Open Source MIPS 32 Simulator (nicht für Performance Simulation geeignet)

Download und weiteres Material: <http://www.cs.wisc.edu/~larus/spim.html>

10.12 Software Tuning

Mit jeder Architekturänderung von OOO-Cores wird es schwieriger die „Performance Bottlenecks“ zu identifizieren und zu bewerten.

TMAM (Top-Down Microarchitecture Analysis Method) geht systematisch vor, um Architekturen zu bewerten und sich auf das „software tuning“ zu konzentrieren.

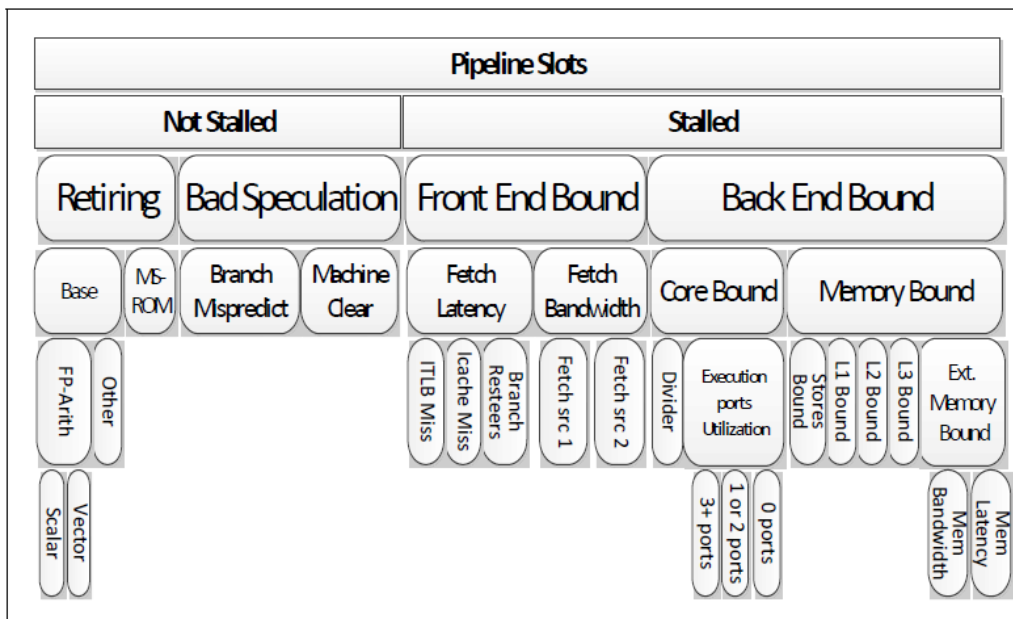


Figure B-1. General TMAM Hierarchy for Out-of-Order Microarchitectures

Quellen:

Internet siehe URLs

Märting: CD-Ergänzungskapitel zum Buch:

Einführung in die Rechnerarchitektur - Prozessoren und Systeme (Hanser 2003)

A.Steiningger: VO Rechnerarchitektur/Performance

Table of Contents

5	Übersicht.....	1
6	Prozessorarchitektur – Überblick.....	1
6.1	Von-Neumann-Architektur / Harvard-Architektur.....	1
6.2	Befehlsabarbeitung.....	1
6.3	Instruktionssatzarchitektur.....	1
6.3.1	Einteilung der Befehlssätze – Anzahl der Adressen.....	1
6.3.2	Einteilung der Befehlssätze – Operanden.....	2
6.3.3	Beschreibung eines Befehls: RTL.....	2
6.3.4	Ausführung eines Befehls.....	4
6.3.5	Adressierung der Operanden – Adressierungsarten.....	5
7	Mikroarchitektur.....	7
7.1	Elemente einer RA.....	7
7.1.1	Bussystem.....	7
7.1.2	Register.....	7
7.1.3	RAM.....	8
7.1.4	ROM.....	9
7.1.5	Rechenwerk.....	9
7.1.6	Steuerwerk.....	9
7.2	CISC - CPU.....	9
7.2.1	Entwicklung Datenpfad CISC-CPU.....	9
7.2.2	Mikroprogrammiertes Leitwerk CISC-CPU.....	10
7.2.3	Befehlsdefinitionen (Mikroprogramm).....	10
8	Eingebettete Systeme (embedded systems).....	11
8.1	Prozessor Funktionsbeschreibung.....	11
8.2	Strukturbeschreibung.....	12
8.3	Entwurf eines Prozessors.....	13
8.3.1	Entwurfsziele.....	13
8.3.2	Vergleich ARM – MIPS.....	13
8.4	Implementierung eines RISC-Prozessors.....	16
8.4.1	Registersatz MIPS.....	16
8.4.2	Befehlsformate.....	17
8.4.3	Überlegungen zur Implementierung.....	18
8.4.4	Teilmenge der implementierten Befehle.....	20
8.4.5	CPU Version 0: R-Befehle.....	21
8.4.6	CPU Version 1: J-Befehl.....	22
8.4.7	CPU Version 2: Immediate.....	23
8.4.8	CPU Version 3: Verzweigung.....	24
8.4.9	CPU Version 4: Load-Store.....	24
8.4.10	CPU Version 5: Sprung.....	25
8.4.11	CPU Version 6: Jump and Link.....	25
8.5	HTL-MIPS.....	26
8.5.1	Befehlsformate.....	26
8.5.2	Registersatz.....	26
8.5.3	MIPS-Assembler-Subset CPU6_15.....	27
8.5.4	HW-Aufbau CPU6_15.....	29
8.5.5	Programmentwicklung mit MARS.....	33
9	Entwicklungsprozesse.....	37
9.1	Geschwindigkeitssteigerung.....	37
9.1.1	Phasenpipeline.....	37
9.1.2	Superskalarität.....	39
9.2	Speicherorganisation.....	41
9.2.1	Cache.....	41
9.2.2	Virtueller Speicher.....	44
9.3	Aktuelle Beispiele für Prozessorarchitekturen.....	51
10	Leistungsbewertung von Ra.....	51
10.1	Leistung.....	51
10.2	Ziele der Leistungsbewertung.....	51
10.3	Wann ist ein Rechner schneller?.....	51
10.3.1	Benutzer eines Arbeitsplatzrechners.....	51
10.3.2	Für Rechenzentrumsleiter.....	51
10.4	Ausführungszeit (execution time).....	52
10.4.1	Wall-clock time, response time, elapsed time.....	52

10.4.2 CPU Time.....	52
10.5 Programmunabhängige Leistungsmodelle.....	52
10.5.1 Taktfrequenz.....	52
10.5.2 MIPS.....	52
10.6 Programmabhängige Leistungsmodelle.....	53
10.6.1 Performance.....	53
10.6.2 relative Performance.....	53
10.6.3 Teiloptimierung.....	53
10.7 Parallelverarbeitung.....	53
10.7.1 Speedup SU.....	53
10.8 Performancesteigerung durch Architekturänderungen.....	54
10.9 Kosten.....	54
10.10 Benchmarks.....	55
10.10.1 Klassen von Benchmarks:.....	55
10.10.2 Whetstone.....	55
10.10.3 Dhrystone.....	55
10.10.4 Linpac.....	55
10.10.5 BAPCo.....	55
10.10.6 SPEC:.....	55
10.10.7 CINT2006.....	55
10.10.8 CFP2006.....	56
10.11 Performance-Analyse und -Simulation.....	56
10.11.1 Monitoring Funktionen und SW-Analyse Werkzeuge.....	56
10.11.2 VTune.....	56
10.11.3 SimpleScalar.....	57
10.11.4 RSIM.....	57
10.11.5 PCSIM.....	57
10.12 Software Tuning.....	57