Kap.3
Mikroarchitektur

Prozessoren,
interne Sicht

- **3.1** elementare Datentypen, Operationen und ihre Realisierung (siehe 2.1)
- **3.2** Mikroprogrammierung
- **3.3** Einfache Implementierung von MIPS
- **3.4** Pipelining

3.2 Mikroprogrammierung

- Realisierung eines Befehlssatzes auf einer Hardware-Struktur
- Befehlsabarbeitung bei CISC und ihre Folgen

Zur Erinnerung: CISC

- Charakterisierung eines CISC Rechners
  - grosser Satz von Maschinenbefehle, zum Teil auch Maschinenbefehle, die recht komplexe Operationen auszuführen haben
  - ➨ sehr unterschiedliche Ausführungszeiten der verschiedenen Maschinenbefehle
  - ➨ Steuerung nicht durch Hardware, sondern durch ein Mikroprogramm, das in einem ROM auf dem Prozessor abgespeichert ist.
- Erster moderner Rechner (IBM 369) war ein CISC Rechner
- **Ein "kleiner" CISC-Rechner zur Illustration**

Maschinen(spiel)sprache (Teil A)

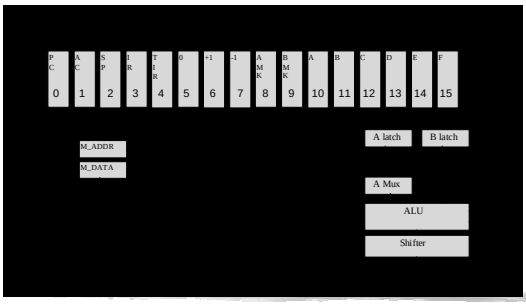
- Drei für den Benutzer sichtbare Register:
  - ac (Akkumulator)
  - sp (Stackpointer)
  - pc (Befehlszähler)
- Die Maschinensprache

0000xxxxxxxxxxxx	LODD	ac:=mem[x]	// 12-Bit Adressenbus
0001xxxxxxxxxxxx	STOD	mem[x]:=ac	
0010xxxxxxxxxxxx	ADDD	ac:=ac+mem[x]	
0011xxxxxxxxxxxx	SUBD	ac:=ac-mem[x]	
0100xxxxxxxxxxxx	JPOS	if ac>0 then pc:=x	
0101xxxxxxxxxxxx	JZER	if ac=0 then pc:=x	
0110xxxxxxxxxxxx	JUMP	pc:=x	
0111xxxxxxxxxxxx	LOCO	ac:=x (0≤x≤4095)	
1000xxxxxxxxxxxx	LODL	ac:=mem[sp+x]	
1001xxxxxxxxxxxx	STOL	mem[x+sp]:=ac	

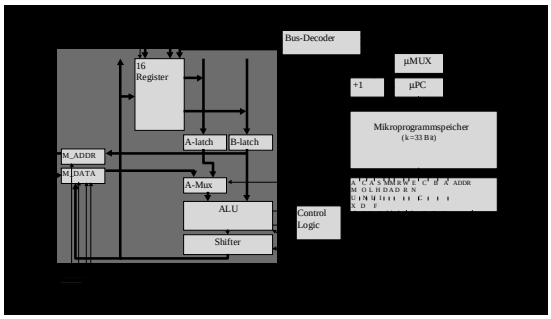
Maschinen(spiel)sprache (Teil B)

1010xxxxxxxxxxxx	ADDL	ac:=ac+mem[sp+x]
1011xxxxxxxxxxxx	SUBL	ac:=ac-mem[sp+x]
1100xxxxxxxxxxxx	JNEG	if ac< 0 then pc:=x;
1101xxxxxxxxxxxx	JNEZ	if ac≠ 0 then pc:=x;
1110xxxxxxxxxxxx	CALL	sp:=sp-1; mem[sp]:=pc; pc:=x;
1111000.....	PSHI	sp:=sp-1; mem[sp]:=m[ac];
1111001.....	POPI	mem[ac]:=mem[sp]; sp:=sp+1;
1111010.....	PUSH	sp:=sp-1; mem[sp]:=ac;
1111011.....	POP	ac:=mem[sp]; ap:=sp+1;
1111100.....	RTN	pc:=mem[sp]; sp:=sp+1;
1111101.....	SWAP	tmp:=sp; sp:=ac; ac:=tmp;
1111110..yyyyyyyy	INSP	sp:=sp+y; (0≤y≤255)
1111111..yyyyyyyy	DESP	sp:=sp-y; (0≤y≤255)

Hardware eines Spielrechners: Datenpfad



Gesamtarchitektur des (Spiel-)Rechners



Aufbau eines Mikroprogrammbefehls

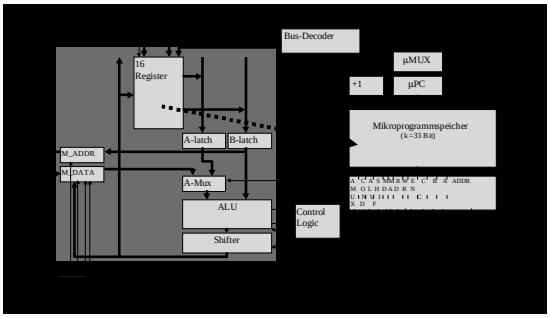
1	**AMUX**	0 → A-Latch;	1 → M_DATA
2-4	**COND**	0 → No jump; 2 → Jump, if Z; 4 → Jump, if Wait	1 → Jump, if N; 3 → Jump
5-6	**ALU**	0 → A+B; 2 → A;	1 → AB; 3 → A'
7-8	**SHIFT**	0 → No shift; 2 → left shift	1 → right shift
9	**MD**	0 → No load;	1 → load
10	**MA**	0 → No load;	1 → load
11	**RD**	0 → No read;	1 → read
12	**WR**	0 → No write;	1 → write
13	**ENC**	0 → disable C;	1 → enable C
14-25	**A, B, C**	determine the active registers	
26-33	**ADDR**	Microprogram jump address	

Zu lösende Arbeit

- Mikroprogramm, das auf unserer Spielhardware jedes Programm, das in der Maschinen(spiel)sprache geschrieben ist, korrekt ausführt.

➪ Mikroprogrammzyklus
- Lade den durch PC adressierten Befehl aus dem Hauptspeicher und schreibe ihn in das Instruktionsregister IR
- Dekodiere den in IR abgespeicherten Befehl
- Lege die Operanden auf den A- bzw. B-Bus (Achtung: es können wait-Zykeln auftreten)
- Führe die verlangte Operation in der ALU und dem Shifter aus
- Speichere das Ergebnis ab.

Wie kann ein Maschinenbefehl dekodiert/ausgeführt werden ?



Lesbare Schreibweise für Mikrobefehle

- Mikroprogramme sind in ihrer eigentlichen Form schwer lesbar

  0 000 00 00 0 0 0 0 1 0001 0001 1010 00000000

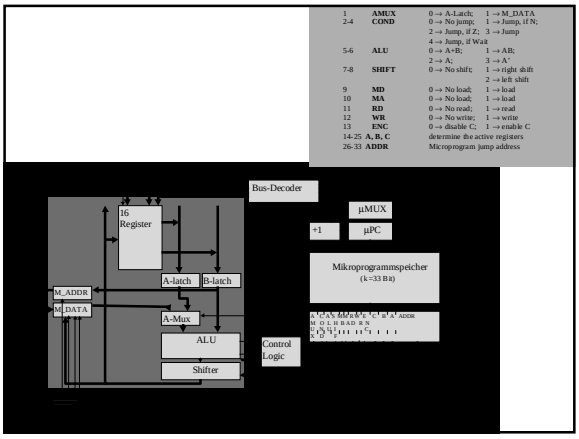
- Mikroinstruktionen sind aber, wie gesehen, in Bereiche eingeteilt

➪ Schreibe im folgenden nur die Teile hin, die verschieden von 0 sind

ENC=1, C=1, B=1, A=10

bzw. verwende eine PASCAL-ähnliche Notation

C:=A+B;



Beispiele für erlaubte "PASCAL"-Notationen

- M_ADDR:=PC; RD // ALU=2; MA=1; RD=1; MD=1;
- RD; // ALU=2; RD=1; MD=1
- IR:=M_DATA // ALU=2; AMUX=1; ENC=1; C=3
- PC:=PC+1 // ENC=1; B=6;
- M_ADDR:=IR; M_DATA:=AC; WR // ALU=2; MD=1; MA=1; WR=1; A=1; B=3;
- ALU:=TIR; If N then goto 15 // ALU=2; COND=1; A=4; ADDR=15

Interpretation des Maschinenbefehls ADDD

Befehl ADDD: 0010xxxxxxxxxxxx ac:=ac+m[x]

Interpretation durch das Mikroprogramm

1. M_ADDR:=PC; RD; // hole den Befehl aus dem Hauptspeicher
2. If WAIT then goto 2; MD:=1; // warten bis Hauptspeicher Datum liefert
3. PC:=PC+1; // erhöhen des Befehlszählers
4. IR:=M_DATA; // schreibe Befehl ins Instruktionsregister
5. // Beginn Dekodierung

...

x. // Ende Dekodierung

x+1. M_ADDR:=TIR; RD // Lese m[x]

x+2. If WAIT then goto x+2; MD:=1; // warten bis Hauptspeicher Datum liefert

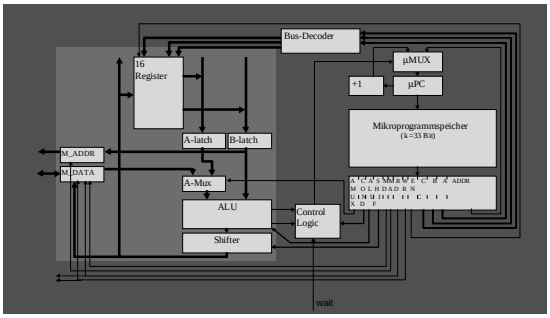
x+3. AC:=AC+M_DATA;

Dekodierung eines Maschinenbefehls

... erfolgt auch durch das Mikroprogramm

1. M_ADDR:=PC; RD; // hole den Befehl aus dem Hauptspeicher
2. If WAIT then goto 2; MD:=1; // warten bis Hauptspeicher Datum liefert
3. PC:=PC+1; // erhöhen des Befehlszählers
4. IR:=M_DATA; if N then goto <label1>;
   // schreibe Befehl ins Instruktionsregister;
   // ist das erste Bit eine 1, so springe
5. TIR:=LSHIFT(IR+IR); if N then goto <label2>;
   // beginnt der Opcode mit 01, so springe
6. TIR:=LSHIFT(TIR); if N then goto <label3>;
   // beginnt der Opcode mit 001, so springe
7. ALU:=TIR; if N then goto <label4>;
   // beginnt der Opcode mit 0001, so springe
8. // Der Opcode ist also 0000, d.h. der Maschinenbefehl ist der ADDD Befehl

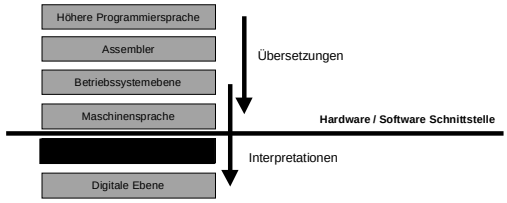
Gesamtarchitektur des (Spiel-)Rechners



Aufbau eines Mikroprogramms

- Enthält die Maschinensprache *n* Instruktionen, so besteht das Mikroprogramm aus *n+1* Teilen:
  - ein Block zum Dekodieren der Befehle
  - zu jeder Maschineninstruktion ein Mikrocode
- Das Mikroprogramm ist im Mikroprogrammspeicher (ROM) vom Hersteller abgelegt und für den Programmierer nicht zugreifbar

CISC/RISC



Überblick

- Einleitung
  - Lit., Motivation, Geschichte, v.Neumann-Modell, VHDL
- Befehlsschnittstelle
- Mikroarchitektur
- Speicherarchitektur
- Ein-/Ausgabe
- Multiprozessorsysteme, ...

Kap.3 Mikroarchitektur

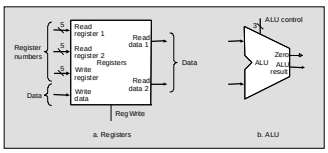
Prozessoren, interne Sicht

- **3.1** Elementare Datentypen, Operationen und ihre Realisierung (siehe 2.1)
- **3.2** Mikroprogrammierung
- **3.3** Einfache Implementierung von MIPS
- **3.4** Pipelining

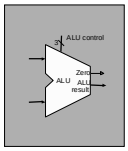
Betrachte der Übersichtlichkeit nur eine Untermenge der MIPS-Sprache:

- Load/Store-Befehle: lw, sw
- Arithmetisch-logische Befehle: add, sub, and, or, slt (auch mit immediate)
- Sprungbefehle: beq, j

Annahme:
die benötigte ALU und Registerbank sind schon implementiert



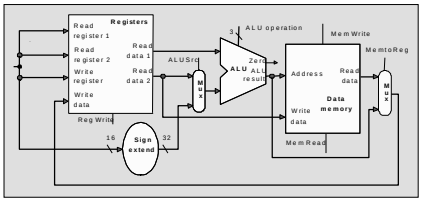
Steuerung der ALU



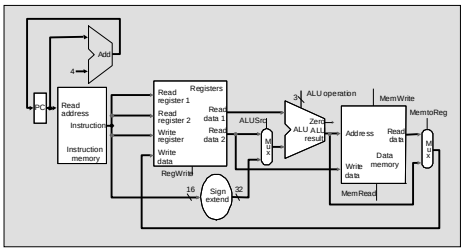
Die ALU kann fünf verschiedene Operationen ausführen, benötigt also 3 Steuerleitungen (**ALUcontrol**)

ALUcontrol	Operation
000	AND
001	OR
010	add
110	substract
111	set on less than

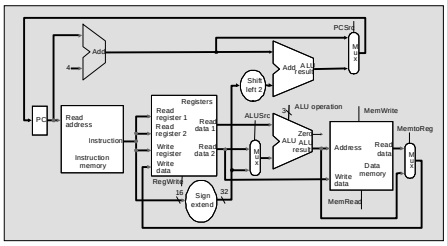
Datenpfad ohne Instruction fetch



Datenpfad mit Inkrementieren von $pc



Datenpfad mit Branch-Logik



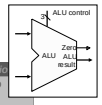
Rolle der ALU im Prozessor

ALU wird nicht nur im Rahmen der arithmetisch-logischen Befehle eingesetzt

- lw Rdest, imm_{16}(Rscr)
  - ➨ Addition zur Berechnung der Adresse
- sw Rsrc1, imm_{16}(Rsrc2)
  - ➨ Addition zur Berechnung der Adresse
- beq Rsrc1, Rsrc2, label
  - ➨ Substraktion zur Berechnung des Vergleiches

	Größe [bit]	6	5	5	5	5	6
Arithmetische Befehle	Format R	op	rs	rt	rd	shamt	funct
Immediate + LOAD/STORE	Format I	op	rs	rt	adr-teil		
Bedingte Sprungbefehle	Format J	op	Sprungadresse				

Bei welchen Instruktionen müssen nun welche Steuersignale gesetzt werden ?

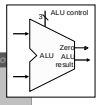


ALUcontrol	Operation
000	AND
001	OR
010	add
110	substract
111	set on less than

ALUOp	Instruction opcode	Instruction operation	Funct field	desired ALU action	ALU control
00	lw	load word	xxxxxx	add	010
00	sw	store word	xxxxxx	add	010
01	beq	branch equal	xxxxxx	substract	110
10	R-type	add	100000	add	010
10	R-type	substract	100010	substract	110
10	R-type	AND	100100	and	000
10	R-type	OR	100101	or	001
10	R-type	set on less than	101010	set less than	111

Ansteuerung der ALU ff

Die Belegung der Steuerleitungen *ALUcontrol* hängen somit nur von *ALUOp* und *Funct* ab.



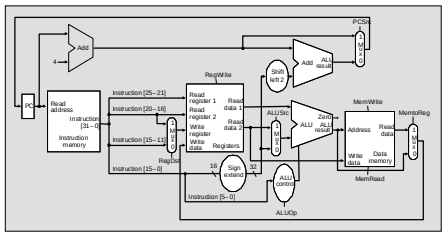
ALUcontrol	Operation
000	AND
001	OR
010	add
110	substract
111	set on less than

ALUOp	Funct field	ALU control
00	xxxxxx	010
00	xxxxxx	010
01	xxxxxx	110
10	100000	010
10	100010	110
10	100100	000
10	100101	001
10	101010	111

ALUOp hängt nur vom Instruktionscode ab

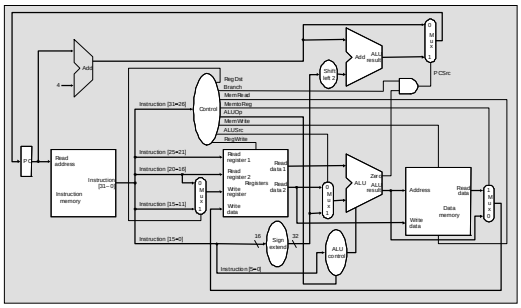
Instruction OPcode	ALUOp control
100011	00
101011	00
000100	01
000000	10

Resultierender Datenpfad



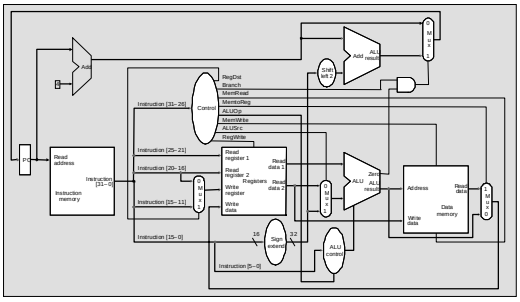
Generierung der Steuersignale

Damit der Datenpfad arbeitet, muss eine Steuereinheit die Steuersignale des Datenpfades in Abhängigkeit der anliegenden Instruktion richtig setzen !



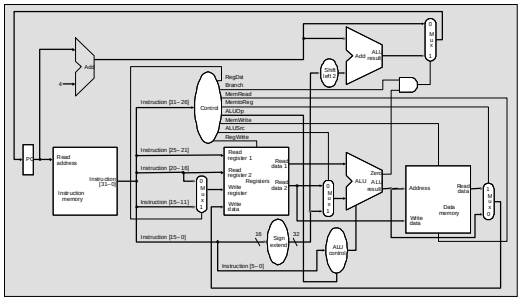
Erster Schritt einer R-type Instruktion

- Holen der Maschineninstruktion
- Erhöhung von $pc um 4



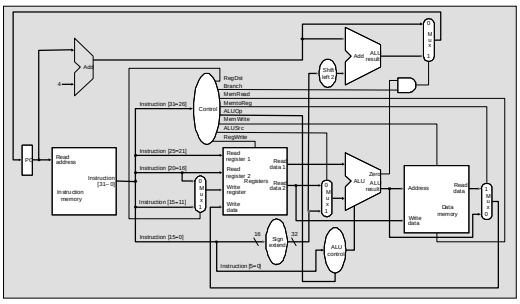
Zweiter Schritt einer R-type Instruktion

- Instruktions-Opcode zur Steuereinheit und Setzen der Steuerleitungen
- Lesen der beiden Register 1 und 2



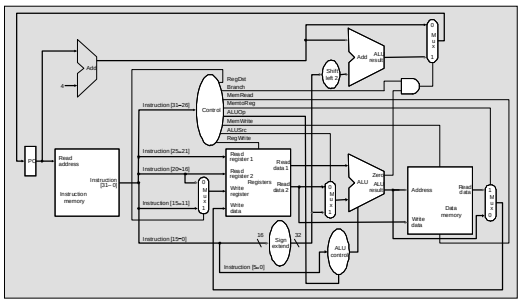
Dritter Schritt einer R-type Instruktion

- Berechnung der ALUcontrol Steuersignale
- Berechnung des Ergebnisses durch die ALU

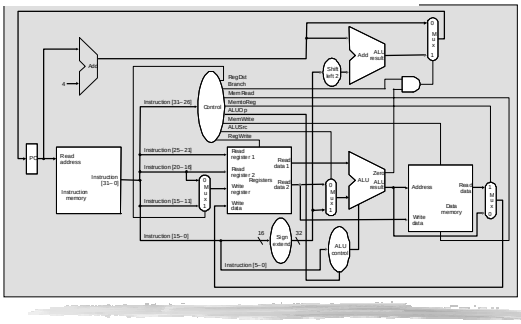


Letzter Schritt einer R-type Instruktion

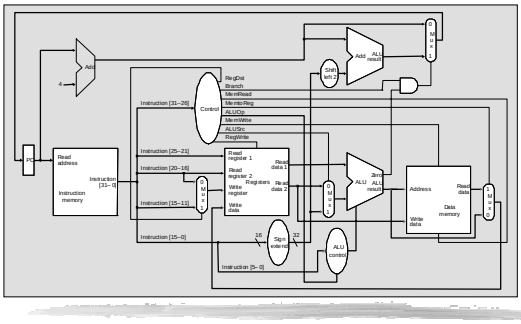
- Zurückschreiben des Ergebnisses in die Registerbank
- Setzen von $pc auf seinen nächsten Wert $pc_{dd}+4$



	Größe [bit]	6	5	5	5	5	6
Arithmetische Befehle	Format R	op	rs	rt	rd	shamt	funct
Immediate + LOAD/STORE	Format I	op	rs	rt	adr-teil		
Bedingte Sprungbefehle	Format J	op	Sprungadresse				

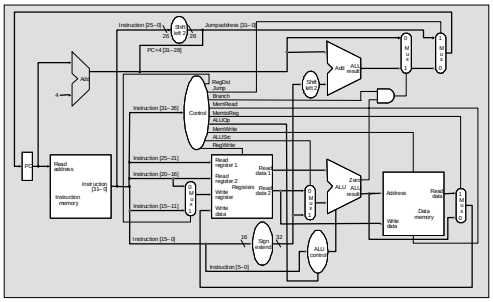


Datenpfad-Operationen bei Branch Equal

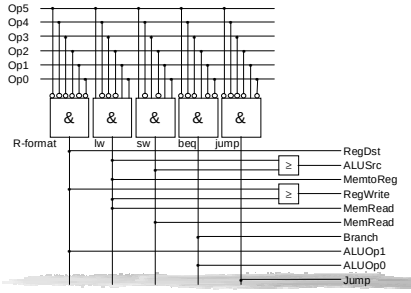


Erweiterung des Datenpfades für J-Instruktion

- In MIPS ergibt sich die Sprungadresse beim Befehl j target durch {($pc+4)[31:28] , instruction[25:0], 2'b00}



HW für die Steuerung



Probleme

- lw benötigt 5 Funktionale Einheiten
  - → lange Gatterlaufzeit
  - → große Taktperiode
  - → alle Instruktionen werden langsamer
- Effekt verschlimmert sich bei noch komplexeren Befehlen (FP-instructions)
- → Mehrtakt Implementierung
  - → Controller wird durch FSM implementiert

FSM-Steuerung