COMPUTER ENGINEERING I

Summary of the Lectures held by Prof. Dr. Thiele

Lukas Cavigelli, July 2011
lukasc@ee.ethz.ch

MIPS ASSEMBLER

REGISTERS

NAME	NMBR	USE	STORE
pc	-	Program Counter	-
hi	-	Special Arithmetic Register	-
lo	-	Special Arithmetic Register	-
$zero	0	Constant Value 0	-
$at	1	Reserved for Assembler	No
$v0-$v1	2-3	Values for Function Results	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$t7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes
$f0-f31	0-31	Floating Point Regs	Yes

ASSEMBLER DIRECTIVES

`.data` [*start_addr*]	Data Segment
`.rodata` [*start_addr*]	Read-Only Data Segment
`.bss` [*start_addr*]	Zero-Initialized Data Segment
`.kdata` [*start_addr*]	Kernel Data Segment
`.ktext` [*start_addr*]	Kernel Text Segment
`.text` [*start_addr*]	Text Segment (actual program)
`.ascii` *"str"*	String in mem., no null-termination
`.asciiz` *"str"*	String in mem., null-terminated
`.byte` $b_1, \dots, b_n$	n successive bytes in mem.
`.double` $d_1, \dots, d_n$	n successive doubles in mem.
`.float` $f_1, \dots, f_n$	n successive floats in mem.
`.half` $h_1, \dots, h_n$	n successive 16-bit quantities in mem.
`.word` $w_1, \dots, w_n$	n successive 32-bit quantities in mem.
`.space` n	Alloc n bytes of space in current seg.
`.extern` *sym size*	*sym* is the name, *size* is in bytes
`.globl` *sym*	Declare label *sym* is global accessable
`.align` n	Align following data to 2^nbyte borders
`.set at`	Make SPIM complain, if `$at` is used
`.set noat`	Make SPIM not compl., if `$at` is used

Use labels as usual to be able to address it.

CORE INSTRUCTION SET PROPERTIES

O: May cause overflow exception
S: SignExtImm = { 16{immediate[15]}, immediate }
Z: ZeroExtImm = { 16{1b'0}, immediate }
B: BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
J: JumpAddr = { PC[31:28], address, 2'b0 }
U: Operands considered usigned

CORE INSTRUCTION SET (32-BIT-WIDE, INCOMPLETE)

Mnemonic	For-mat	Comment	Operation (Verilog)	Properties					
				O	S	Z	B	J	U
`add`	R	Add	R[rd] = R[rs] + R[rt]	x					
`addi`	I	Add Immediate	R[rt] = R[rs] + SignExtImm	x	x				
`addiu`	I	Add Imm. Unsigned	R[rt] = R[rs] + SignExtImm		x				x
`addu`	R	Add Unsigned	R[rd] = R[rs] + R[rt]		x				x
`sub`	R	Subtract	R[rd] = R[rs] - R[rt]	x					
`subu`	R	Subtract Unsigned	R[rd] = R[rs] - R[rt]						x
`and`	R	And	R[rd] = R[rs] & R[rt]						
`andi`	I	And Immediate	R[rt] = R[rs] & ZeroExtImm			x			
`nor`	R	Nor	R[rd] = ~(R[rs] | R[rt])						
`or`	R	Or	R[rd] = R[rs] | R[rt]						
`ori`	I	Or Immediate	R[rt] = R[rs] | ZeroExtImm			x			
`xor`	R	Xor	R[rd] = R[rs] ^ R[rt]						
`xori`	I	Xor Immediate	R[rt] = R[rs] ^ ZeroExtImm						
`sll`	R	Shift Logical Left	R[rd] = R[rs] << shamt						
`srl`	R	Shift Logical Right	R[rd] = R[rs] >> shamt						
`sra`	R	Shift Arithm. Right	R[rd] = R[rs] >>> shamt						
`sllv`	R	Shift Logic. Left Var.	R[rd] = R[rs] << R[rt]						
`srlv`	R	Shift Logic. Right Var.	R[rd] = R[rs] >> R[rt]						
`srav`	R	Shift Arith. Right Var.	R[rd] = R[rs] >>> R[rt]						
`slt`	R	Set Less Than	R[rd] = (R[rs] < R[rt]) ? 1 : 0						
`slti`	I	Set Less Than Imm.	R[rt] = (R[rs] < SignExtImm) ? 1 : 0		x				
`sltiu`	I	Set Less Than Imm. Unsign.	R[rt] = (R[rs] < SignExtImm) ? 1 : 0		x				x
`sltu`	R	Set Less Than Unsign.	R[rd] = (R[rs] < R[rt]) ? 1 : 0						x
`beq`	I	Branch On Equal	if (R[rs] == R[rt]) PC = PC + 4 + BranchAddr				x		
`bne`	I	Branch On Not Equal	if (R[rs] != R[rt]) PC = PC + 4 + BranchAddr				x		
`blt`	P	Branch Less Than	if (R[rs] < R[rt]) PC = PC + 4 + BranchAddr						
`bgt`	P	Branch Greater Than	if (R[rs] > R[rt]) PC = PC + 4 + BranchAddr						
`ble`	P	Branch Less Than Or Equal	if (R[rs] <= R[rt]) PC = PC + 4 + BranchAddr						
`bge`	P	Branch Greater Than Or Eq.	if (R[rs] >= R[rt]) PC = PC + 4 + BranchAddr						
`j`	j	Jump	PC = JumpAddr					x	
`jal`	J	Jump And Link	R[31] = PC + 4; PC = JumpAddr					x	
`jr`	R	Jump Register	PC = R[rs]						
`jalr`	R	Jump And Link Register	R[31] = PC + 4; PC = R[rs]						
`move`	P	Move / Copy	R[rd] = R[rs]						
`lb`	I	Load Byte	R[rt] = { 24'b0, M[R[rs]+ZeroExtImm](7:0) }			x			
`lbu`	I	Load Byte Unsigned	R[rt] = { 24'b0, M[R[rs]+ZeroExtImm](7:0) }		x				
`lh`	I	Load Halfword	R[rt] = { 16'b0, M[R[rs]+ZeroExtImm](15:0) }			x			
`lhu`	I	Load Halfword Unsigned	R[rt] = { 16'b0, M[R[rs]+ZeroExtImm](15:0) }		x				
`lui`	I	Load Upper Immediate	R[rt] = { imm, 16'b0 }						
`lw`	I	Load Word	R[rt] = M[R[rs]+SignExtImm]		x				
`li`	P	Load Immediate	R[rd] = immediate						
`la`	P	Load Address	R[rd] = immediate						
`sb`	I	Store Byte	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)		x				
`sh`	I	Store Halfword	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)		x				
`sw`	I	Store Word	M[R[rs]+SignExtImm] = R[rt]		x				
`div`	R	Divide	Lo = R[rs] / R[rt]; Hi = R[rs] % R[rt]						
`divu`	R	Divide Unsigned	Lo = R[rs] / R[rt]; Hi = R[rs] % R[rt]						x
`mult`	R	Multiply	{Hi, Lo} = R[rs] * R[rt]						
`multu`	R	Multiply Unsigned	{Hi, Lo} = R[rs] * R[rt]						x
`bc1t`	FI	Branch On FP True	if (FPCond) PC = PC + 4 + BranchAddr				x		
`bc1f`	FR	Branch On FP False	if (!FPCond) PC = PC + 4 + BranchAddr				x		
`c.x.s*`	FR	FP Compare Single	FPCond = (F[fs] *op* F[ft]) ? 1 : 0						
`c.x.d*`	FR	FP Compare Double	FPCond = ({F[fs],F[fs+1]} *op* {F[ft],F[ft+1]})?1:0 *(*x is `eq`, `lt` or `le`)(*op* is ==, < or <=)						
`add.[d/s]`	FR	FP Add [double/single]	F[fd] = F[fs] + F[ft], double-version too long						
`div.[d/s]`	FR	FP Divide [double/single]	F[fd] = F[fs] / F[ft]						
`mul.[d/s]`	FR	FP Multiply [double/single]	F[fd] = F[fs] * F[ft]						
`sub.[d/s]`	FR	FP Subtract [double/single]	F[fd] = F[fs] – F[ft]						
`mfhi/mflo`	R	Move From Hi / Lo	R[rd] = Hi / R[rd] = Lo						
`mfc0/mtc0`	R	Move From / To Coproc 0	R[rd] = CR[rs] / CR[rs] = R[rd]						
`lwc1`	I	Load FP Single	F[rt] = M[R[rs] + SignExtImm]		x				
`ldc1`	I	Load FP Double	F[rt] = M[R[rs] + SignExtImm]; … too long		x				
`swc1`	I	Store FP Single	M[R[rs] + SignExtImm] = F[rt]		x				
`sdc1`	I	Store FP Double	M[R[rs] + SignExtImm] = F[rt]; … too long		x				

MIPS SAMPLES

```
sll $rd, $rt, shiftamt
sub $rd, $rs, $rt
addi $sp, $sp, -8
beq $s1, $s2, [offset+1]
beq $q1, $s2, -1 #endless loop
lw $15, 4($2)     #address = $2+4 (Bytes)
big-endian: most signific. byte at lowest address
synchronisation: ll und sc
```

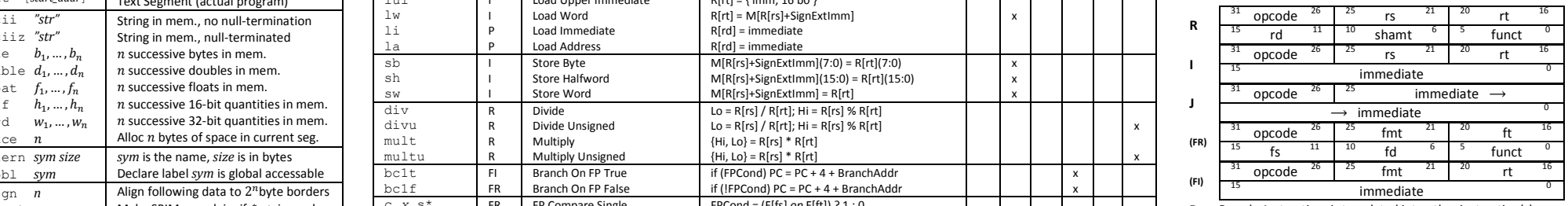
SYSCALLS

Service	$v0	Arguments	Result
print_int	1	integer $a0	
print_float	2	float $f12	
print_double	3	double $f12/$f13	
print_string	4	string $a0	
read_int	5		integer $v0
read_float	6		float $f0
read_double	7		double $f0
read_string	8	buf $a0, buflen $a1	
sbrk	9	amount $a	address $v0
exit	10		

EXCEPTION CODES

Num	Name	Cause of Exception
0	Int	Interrupt (hardware)
4	AdEL	Address Error (load or instruction fetch)
5	AdES	Address Error (store)
6	IBE	Bus Error on Instruction Fetch
7	DBE	Bus Error on Load or Store
8	Sys	Syscall Exception
9	Bp	Breakpoint Exception
10	RI	Reserved Instruction Exception
11	CpU	Coprozessor Unimplemented
12	Ov	Arithmetic Overflow Exception
13	Tr	Trap
15	FPE	Floating Point Exception

BASIC INSTRUCTION FORMATS



P Pseudo-Instruction: is translated into other instruction(s)
(FR) and **(FI)** are floating point instructions (not treated in lecture)

TIPPS, TRICKS & COMMON MISTAKES

- convert for to while and use pointer arithmetics
- after using mult, get result with mflo
- when doing pointer arith. you might want to use +4 not +1
- subi does not exists, use addi with negative immediate
- correct loading: lw t0,0(t1), do not forget brackets and 0
- work from inner to outer loops, assign variables to registers

INTRODUCTION

Embedded Systems: Hidden part of a whole system.
- real-time capable
- specialized: optimized, not user-programmable
- reliable: high availability
- efficient: energy, size, weight, cost

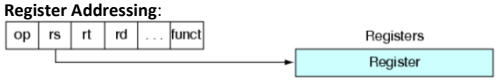
INSTRUCTION SET

Layer Model:
C program → [Compiler] → assembly program → [Assembler] → program object code (machine language) + object code from library → [Linker] → Executable → [Loader] → Memory
Neumann-Cycle:
Load instructions from memory → decode instructions → fetch operands (from memory or registers) → execute instruction → save result → identify next instruction → [start over]

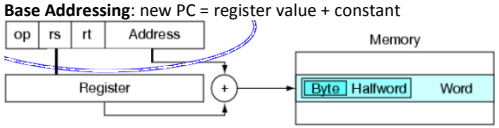
ADDRESSING METHODS

Direct Addressing (also: immediate addressing):

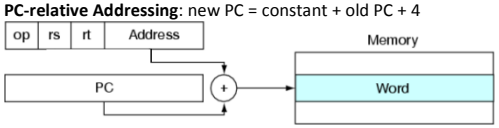
Register Addressing:



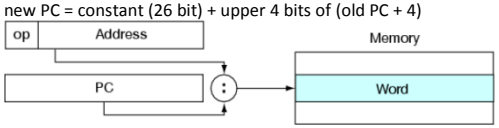
Base Addressing: new PC = register value + constant



PC-relative Addressing: new PC = constant + old PC + 4



Pseudo-direct Addressing:
new PC = constant (26 bit) + upper 4 bits of (old PC + 4)



IMPORTANT INSTRUCTIONS

lw $t0, 100($s2) # $t0 = Memory[100+$s2]
sw $t0, 100($s2) # byte-wise addressing
slti $t1, $s2, 100 # if($s2<100) then $t1=1 else $t1=0
slt $t1, $s2, $s3 # if($s2<$s3) then $t1=1 else $t1=0

blbabla

JUMPING & BRANCHING

- jump instr. (j, jal) are pseudo-direct of register addressing
  - jal label1 # $ra=PC+4, go to label1
  - j label1 # go to label1 (set PC=label1*4)
  - jr $ra # set PC=$ra
- branch instructions (beq, bne) are pc-relative
  beq $s1, $s2, imm # if($s1==$s2) then PC=PC+4*(imm+1)

DATA FORMAT

Data Formats: byte (8 bit), half word (16 bit), word (32 bit)
Endianness: big endian is opposite of little endian. Big endian: Most significant *byte* of a word is at its lowest address. A word is addressed with the byte address of its most significant *byte*. Later instructions are at a higher address.
Unsigned integer: $B = \sum_{i=0}^{n-1} b_i 2^i$
Extension $n \rightarrow m$: $B = \sum_{i=n}^{m-1} 0 \cdot 2^i + \sum_{i=0}^{n-1} b_i 2^i$
Signed Integer (two's complement):
$B = -b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i$
Extension: $B = -b_{n-1}2^{m-1} + \sum_{i=n}^{m-2} b_{n-1}2^i + \sum_{i=0}^{n-1} b_i 2^i$
Determine sign: $(B < 0) \Leftrightarrow (b_{n-1} = 1)$
Negation: $-B = -(1 - b_{n-1})2^{n-1} + \sum_{i=0}^{n-2}(1 - b_i)2^i + 1$

SYNCHRONIZATION

Special Instructions:

```
ll    $t1,offset($s1) # load linked
sc    $t0,offset($s1) # store conditional
```

Usage: (Example)

```
try:   addi $t0, $zero, 1      # $t0 = 1
       ll $t1, 0($s1)          # ll on lock var
       bne $t1, $zero, try     # if lock!=0 -> try
       sc $t0, 0($s1)          # sc on lock var
       beq $t0, $zero, try     # if sc fails -> try
       …                       # use memory
       sw $zero, 0($s1)        # lock = 0
exit:  …
```

ASSEMBLER & MACHINE PROGRAM

Assembler: Translates the assembler prog. to a machine prog. The assembler program contains: comments, symbolic opcodes, symbolic register names, symbolic marks (line marks), macros The assembler also handles pseudo instructions and latencies.
Pseudo-Instruction: move $t0, $t1 → add $8, $0, $9
Latency: ***load operations*** (e.g. lw) are only available in the second instructionafter the load operation. This is handled.
branch delay slot: the first instruction after a branch operation (e.g. bne) is always executed. This is handled by the assembler.

ASSEMBLER

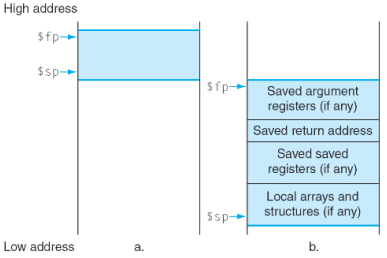
BRANCHING

while-loop:

```
        while (save[i] == k) i = i + 1;
        $s3 = i, $s5 = k, $s6 = &save[0]
loop:   sll   $t1, $s3, 2      # $t1 = 4 * i
        add   $t1, $t1, $s6
        lw    $t0, 0($t1)
        bne   $t0, $s5, exit
        addi  $s3, $s3, 1
        j     loop
exit:   …
```

FUNCTIONS

Context of a subprogram('activation record','procedure frame'): arguments, register contents, local variables
Function call: [caller]: 1: Save temporary registers (transfer values from $a0-$a3, $t0-$t9, $v0-$v1 to stack) → 2: put arguments into $a0-$a3 (and on the stack, if needed) → 3: Jump-instruction (jal)
[callee]: 4: Allocate frame on stack (decrease $sp) → 5: Store saved registers, if needed ($fp, $ra, $s0-$s7) → 6: set frame pointer $fp → 7: store results in $v0, $v1 → 8: restore saved registers → 9: deallocate frame → 10: jump back (jr $ra)
[caller]: 11: restore temporary registers
Function call: jal funcLbl
Function return: jr $ra
Temporary registers have to be saved by the calling program.
Stored registers have to be saved by the called program
Memory Arrangement: a: before, b: after



Example Code:

```
void swap(int v[], int k) {
  int temp; temp = v[k]; v[k] = v[k+1]; v[k+1] = temp; }
```

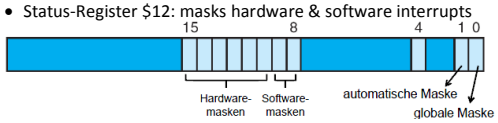
steps 1 to 3?!

```
swap: addi $sp,$sp,-8 # 4: zwei Register werden gesichert
      sw $fp,4($sp)   # 5: fp wird in swap geändert
      sw $s0,0($sp)   # 5: $s0 wird in swap geändert
      addi $fp,$sp,4  # 6: fp zeigt nun auf Rahmenanfang
      sll $t6,$a1,2   #    $t6 = k * 4
      add $t6,$a0,$t6 #    $t6 = v + (k * 4)
      lw $t7,0($t6)   #    temp = v[k]
      lw $s0,4($t6)   #    $s0 = v[k+1]
      sw $t7,4($t6)   #    v[k+1] = temp
      sw $s0,0($t6)   #    v[k] = $s0
      addi $sp,$fp,-4 # 8: Vorbereitung für rückspeichern
      lw $fp,4($sp)   # 8: Rückspeichern von fp
      lw $s0,0($sp)   # 8: Rückspeichern von $s0
      addi $sp,$sp,8  # 9: Rahmen de-allozieren
      jr $ra          # 10: Zurückspringen
```

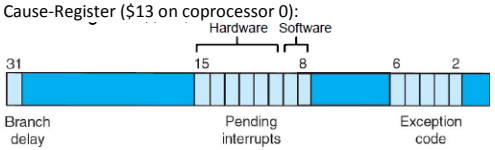
step 11

INTERRUPTS

Reasons: interrupt signal, arithmetic except. (e.g. zero division), address error, bus error, software (break, syscall), debug, timer
Implementation: 3 registers:
- Status-Register $12: masks hardware & software interrupts



- Cause-Register ($13 on coprocessor 0):



- EPC-Register ($14 on coprocessor 0)

Flow: interrupt routineis executed, if an interrupt occurs, corresponding hardware or software mask in $12 is set, global mask ist set in $12 and the automatic mask is set in $12.
- programm execution is stopped, PC is stored in EPC register and set to 0x80000180. There we find a user defined interrupt handler.
- Further interrupts are prevented through the automatic mask.
- the cause register is set (pending interrupts, exception code)
- interrupt handler is executed

The left program has not saved any registers (!)
Jump back instruction: rti
Example – Enable Interrupts:

```
li      $t4, 2
sw      $t4, 0($t0)     # set bit 1 of receiver control register
li      $t4, 0x0801     # create bitmask for interrupt status
mtc0    $t4, $12        # write status to status reg of coproc
```

Example – Interrupt Routine:

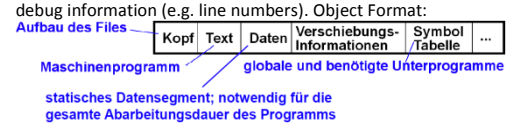
```
.ktext  0x80000180      # forces interrupt routine to 0x80000180
set noat                # tell assembler to stop using $at
sw      $at, save_at    # now we can safely use it
.set at                 # give back $at to the assembler
sw      $t0, save_t0    # save registers (also the t registers)
        do whatever you want
mtc0    $0, $13         # clear cause register
lw      $t0, save_t0    # restore registers
.set noat
lw $at, save_at         # restore $at register
.set at
eret                    # exception return
.kdata                  # alloc memory to save variables
save_at: .word 0
save_t0: .word 0
```

CISC V. RISC, IA-32

RISC: simple encoding, all instructions have the same size, suitable for high frequency & parallelization, many usable registers.
IA-32: The Pentium architecture translates IA-32 instr. to RISC.

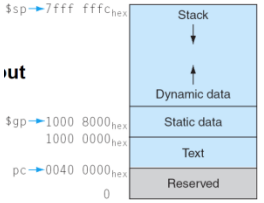
PROGRAM TO EXECUTION

Flow overview: high level language → [compiler] → assembler program → [assembler] → object program + libraries → [Linker] → machine program → [Loader] → machine program in memory
High level language: short, readable, maintainabe, architecture independent, can be problem specific (matlab, lisp, …)
Assembler Language: architecture specific, symbolic representation of machine program, contains directives for translation to object code, contains comments, symboli opcodes, symbolic register names, symbolic labels, macros.
Assembler: translates opcodes, register names and labels. Creates a list of non-resolved global labels and references.
Object Program: contains executable code, symbols (mapping of function & variable names to addresses), data (init values, size, addresses, … of global variables & constants), references to data & functions in other object-files, relocation information, debug information (e.g. line numbers). Object Format:



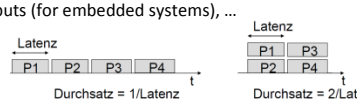
Examples on page 4-11.

Linker: Links all the program parts together, leaving no unresolved labels. Includes all the necessary references to libraries. Sets the memory spaces for program segments, …
Example: page 4 – 14
Loader: Defines the size and address range of text and data segment. Copies program text and data into address range. Puts arguments of the progam on the stack, initializes registers.
Typical memory layout:

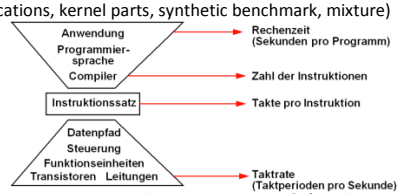


PERFORMANCE

Evaluating a computer: cost, energy consumption, execution time (latency, cpu time), throughput, response time on interrputs (for embedded systems), …



Performance Assessment: frequency, CPI (avg. cycles per instr.), MIPS (millions instructions per second), MFLOPS (millions floating point operations per second), Benchmarks (real applications, kernel parts, synthetic benchmark, mixture)



$$\text{CPU-Time} = \frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Time}}{\text{Cycle}}$$

$$\text{IPS} = \frac{\text{Instructions}}{\text{Time}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{1}{\text{CPU-Time}}$$

$$\text{CPI} = \sum_{i=1}^{n} \text{CPI}_i \cdot F_i, \quad F_i = N_i / N$$

with n: # different instructions, CPI_i: cycles for instruction type i
F_i: fraction of instruction i of all instructions of the program
N_i: # instr. of type i in the program, N: #instr. in the program

INFLUENCE ON PERFORMANCE BY LEVEL

	# instr.	CPI	Frequency
algorithm	x	x	
compiler	x	x	
instruction set	x	x	x
architecture		x	x
technology			x

Criteria for the instruction set: efficient translation possible, few instructions per program, efficient implementation, few cycles per instruction, resources used where useful (large F_i)

SPEC-Benchmark: $SPEC = \sqrt[n]{\prod_{i=1}^{n} \frac{\text{CPU-Time}_i}{\text{Reference-Time}_i}}$, i: program id
SPECINTC2006 (CPU, no I/O): List of programs on page 5 – 11.

INPUT & OUTPUT

Transaction-I/O: lots of small amounts of data, frequent access
File-I/O: large amounts of data per time
Event-I/O: short reaction-time, as many events should be processed per time
I/O data rate or **bandwidth**: amount of data per time
I/O response time: overall time for a single I/O operation

BUSSES

Bus: Common used communication link
- pro: new units can be added easily, cheap
- contra: bottle neck of communication, throughput limited by bus length and # devices, has to support very different devices

Bus organization:
- control lines: request, acknowledgement, reservation, …
- data lines: data, addresses, complex control information, …

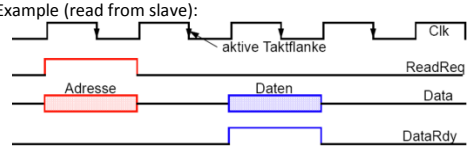
I/O-Transaction: reserve bus, send address, send or receive data, release bus
Master/Slave:
Master: starts and ends bus transaction, sends the address.
Slave: responds to request and address, sends/receives data.
Types of busses:
- CPU-Memory-Bus: short, high speed
- I/O-Bus: has to serve many different units
- AGP-Bus, PCI-Bus, …

Design Decisions:

Option	Hohe Leistung	Geringe Kosten
Synchronisation	synchron (vorausgesetzt, die Taktverschiebung ist klein)	asynchron
Busbreite	separate Adress- und Datenleitungen	multiplexen von Adress- und Datenleitungen
Datenbreite	mehrere Bytes können pro Buszyklus übertragen werden	Enger ist billiger
Blockgrösse	Übertragung mehrerer Worte pro Transaktion reduziert Verluste durch Protokoll	Einzelwortübertragung ist billiger
Busmaster	mehrere Master (erfordert Arbitrierung)	ein Master (keine Arbitrierung erforderlich)
geteilte Übertragung	getrennte Anfordungs- und Übertragungstransaktionen (erfordert Arbitrierung)	Anforderung und Übertragung in einer Transaktion

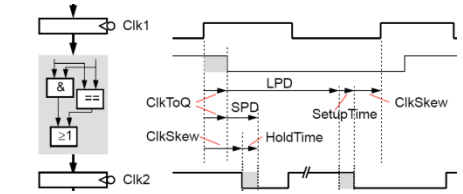
SYNCHRONOUS PROTOCOLS

Synchronous: Common clock line for synchronisations, protocol relative to this clock. *Pro*: fast, if clock skew small.
Con: every unit has to support the same frequency
Example (read from slave):

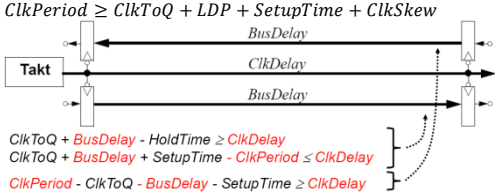


Timing:

ClktoQ	delay from rising clock edge to valid reg output
ClkSkew	delay of the clock signal between registers
ClkPeriod	period of the clock
SetupTime /HoldTime	time the input signal has to be valid before/after rising clock edge
SPD/LPD	shortest/longest delay between output and input of all registers



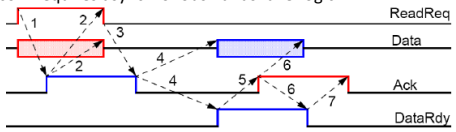
$$ClkToQ + SPD \geq ClkSkew + HoldTime$$
$$ClkPeriod \geq ClkToQ + LDP + SetupTime + ClkSkew$$



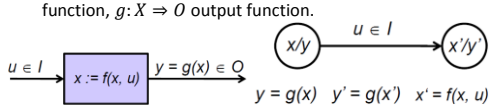
Possible Improvements:
- generate clock at data source (same delay as data) (SDRAM)
- clock is lead back to clock generator. Each unit reconstructs the original clock signal (RAMBUS)

ASYNCHRONOUS PROTOCOLS

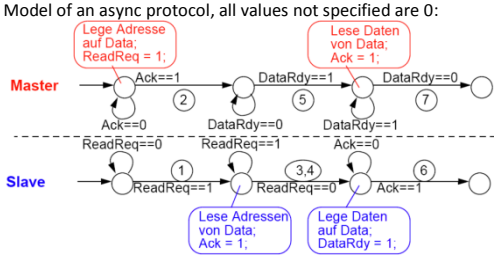
Properties:
- no clock line, 'handshake'-protocoll
- pro: no common clock, delay independent function, support of heterogenous units
- con: requires asynchronous handshake-logic



Specification using a Finite State Machine (FSM):
A deterministic finite state machine is a 6-tupel (I, O, X, x_0, f, g) with I: finite input set, O: finite output set, X: finite state set and $x_0 \in X$ init state, $f: X \times I \Rightarrow X$ state transition function, $g: X \Rightarrow O$ output function.

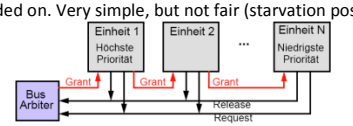


Model of an async protocol, all values not specified are 0:

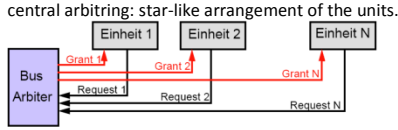


Transmission mechanisms:
- *block transmission*: multiple words (1 block) are transmitted per transaction, address is only sent once, bus blocked until last word is transmitted.
- *split transaction*: bus transmits small data chunks (e.g.words)

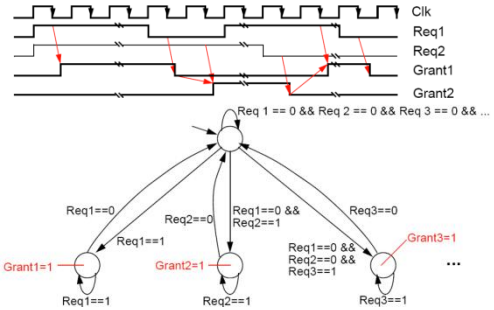
Arbiter Mechanisms:
One ***Busmaster*** control all bus accesses: initiates and controls all bus reservations, slave responds to read & write requests, master (e.g. CPU) takes part in all transactions
Use of ***Arbiter Schemes***: A busmaster signals a bus request. It cannot use the bus until access is granted. The busmaster signals the end of its transaction.
Arbiter mechanisms:
- distributed arbitring by self selection: Each unit sets identification number on the bus → unit with highest priority can access the bus.
- distributed aritring through collision detection: unit signals reservation, is bus is empty, otherwise it waits.
- daisy chain: Arranging the unit in a priority chain. Grant-signal is handed on. Very simple, but not fair (starvation possible)



- central arbitring: star-like arrangement of the units.



Protokoll eines synchronen Arbiters:



OPERATING SYSTEM

The operating system is the interface between the IO-hardware and the user programm requesting a transaction.
I/O-System: is used by different systems (resource conflicts), uses interrupts, is handled by lower levels of the OS
Tasks of the OS: protection of resources, abstraction for programms, mutual exclusion of the users, fair access for all users.
Ways of communication:
- Instructions (OS → IO): ***Addressing***: ***memory mapped I/O*** (like memory address, but instead of memory a device)
- Messages (IO → OS): ***Polling***: periodical sampling of the status registers, ***Interrupt***: Unit interrupts current programm (special hardware), usage of cause/status-registers
- Data (OS ↔ IO): needs a lot of performance → ***DMA***-Function: outside of the CPU, is bus master, transactions CPU-independent. DMA has direct memory access.

Storage hierarchy:
Register → Cache → Main Memory → Disk Memory
Access time to a hard drive:

Access time = search time + rotation latency + transmission time + controller latency + queue delay

More on hard drives, RAID, … → TIK II summary

CPU – SINGLE-CYCLE IMPLEMENTATION

Data Path: processing and transportation of instructions and data, supports all operations and transports

Control Path: processing and transportation of control data. Hardware interprets instructions, e.g. as micro programming language

MIPS subset:

- R-Typ Instruktionen:

add	add rd, rs, rt	000000	rs	rt	rd	00000	100000
subtract	sub rd, rs, rt	000000	rs	rt	rd	00000	100010
AND	and rd, rs, rt	000000	rs	rt	rd	00000	100100
OR	or rd, rs, rt	000000	rs	rt	rd	00000	100101
set less than	slt rd, rs, rt	000000	rs	rt	rd	00000	101010

- Speicherinstruktionen

load word	lw rt, imm(rs)	100011	rs	rt	imm
store word	sw rt, imm(rs)	101011	rs	rt	imm

- Verzweigungsinstruktionen

branch on equal	beq rs, rt, imm	000100	rs	rt	imm
jump	j target	000010	target		

Elementary Operations:

- Der Prozessor stellt die folgenden Elementaroperationen und Variablen zur Verfügung:
  - *Zwischenvariablen* zum Speichern von 32 Bit Daten und Instruktionen: `A, B, ALUOut, Target, PC` (program counter), `NPC` (next program counter), `IR` (instruction)
  - *Interne Register* des Prozessors: `Reg[i], 0 ≤ i ≤ 31`
  - *Hauptspeicher* an der Adresse i: `Mem[i]`
  - *Verschiebung* eines Wortes um 2 Bit nach links und auffüllen mit `'00': '01110' << 2 = '0111000'`
  - *Aneinanderhängen* von Bits: `'001' <> '110' = '001110'`
  - *Erweiterung* eines Halbwortes auf ein Wort mit/ohne Vorzeichenerweiterung: `SignExt(), ZeroExt()`
  - *Arithmetische Operation* wobei *op* ∈ {'add', 'subtract', 'AND', 'OR', 'setOnLessThan'}: `ALU(a, b, op)`

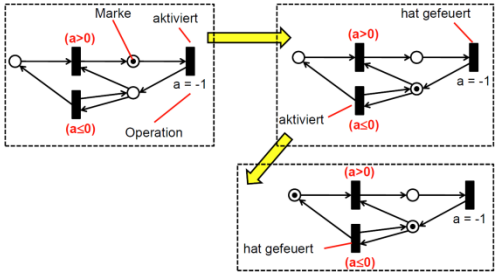
CONTROL FLOW & PETRI NETS

Control Flow Graph (Petri nets):

- Nodes: operations to be executed
  - flow node: activated, if there is a marking on each input edge. On sparking a marking disappears on each input edge and a new one appears at each ouput edge.
  - connection node: activated, if *at least one* of the input edges has a marking. On sparking for each a input marking disappears and a new one appears at each output edge.
  - branching node: activated, if there is a marking on each input edge. On sparking a marking is removed on each input edge and a marking is added to the *selected* output edge (e.g. if-branching)
- Edges: direction of the control flow
- Markings: current state

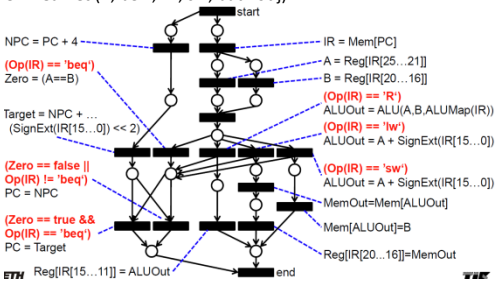
blabla schooner formulieren, seite 8-9 ff

Example:

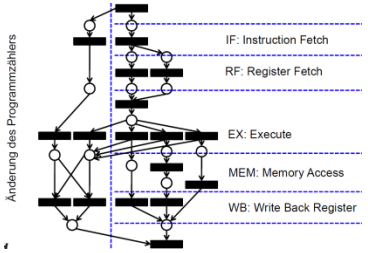


Examples for R, beq, j, lw, sw on pages 8-11 ff, all on one: p.8-18

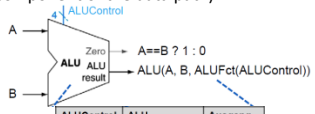
Unified net (R, bew, lw, sw, but not j):



Principal flow:

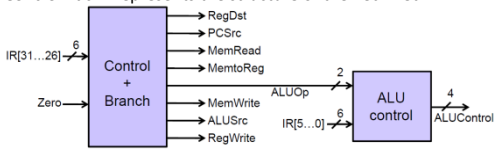


ALU: (as a component of the data path)



ALUControl Eingang	ALU Funktion	Ausgang
0010	add	A+B
0110	sub	A-B
0000	AND	A && B
0001	OR	A || B
0111	setOnLessThan	A<B ? 1 : 0

Control Path: Represents the structure of the Petri net:



ALU Control & Control Units:

Control ALU Control

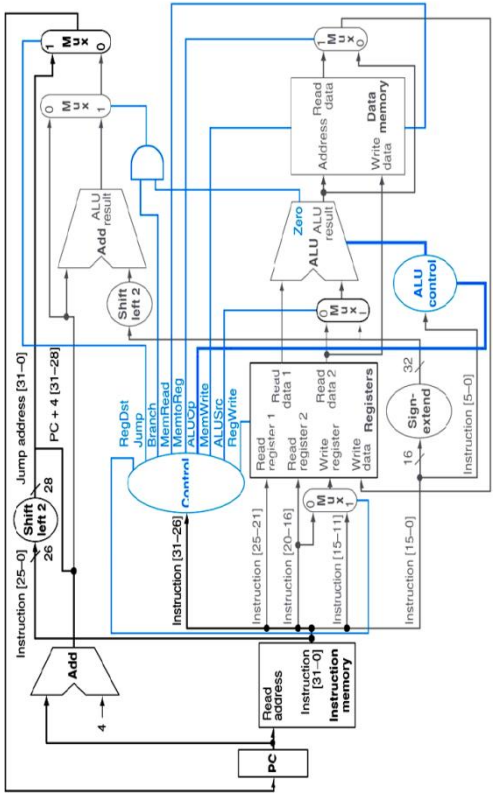
Operation	opcode IR[31…26]	ALUOp	functcode IR[5…0]	ALU Funktion	ALUControl
load word (lw)	100011	00	XXXXXX	add	0010
store word (sw)	101011	00	XXXXXX	add	0010
branch equal (beq)	000100	01	XXXXXX	subtract	0110
add (add)	000000	10	100000	add	0010
subtract (sub)			100010	subtract	0110
and (and)			100100	AND	0000
or (or)			100101	OR	0001
set-on-less-than (slt)			101010	setOnLessThan	0111

Control Unit:

Instruktion	IR[31…26]	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp
R-Typ	000000	1	0	0	1	0	0	0	10
lw	100011	0	1	1	1	1	0	0	00
sw	101011	X	1	X	0	0	1	0	00
beq	000100	X	0	X	0	0	0	1	01

Eingang — Ausgangssignale

Entire single-cycle implementation: (with j)



Parallel instruction Processing:

- Single Cycle: all instructions are processed in one clock cycle. → if long instructions have to be implemented, the clock has to be very slow
- Multiple Cycle: splitting instructions into multiple segments, which need one clock cycle each. → faster clock, but more registers necessary

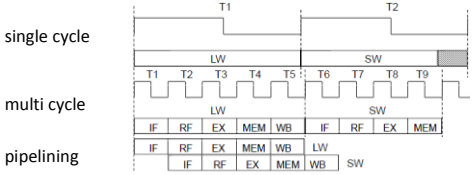
Implementation: Control flow graph → Finite State Machine → explicit function to get to the next state → PLA (chap. 7)

PIPELINING

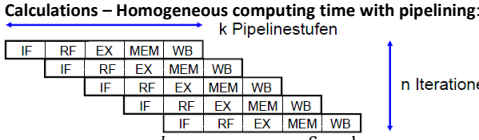
Goal: run multiple instructions in parallel

Breakdown of the instructions into 5 **phases**:
IF (instruction fetch), ID (instruction decode),
EX (execute), MEM (memory), WB (write back)

Comparison of different architectures:



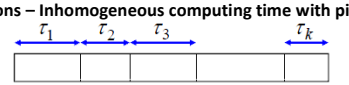
Calculations – Homogeneous computing time with pipelining:



$$\text{Speedup} = \frac{n \cdot k}{k + n - 1}, \quad \text{Efficiency} = \frac{\text{Speedup}}{k} = \frac{n}{k + n - 1}$$

k: pipelining stages (here 5), n: parallel instructions/iterations

Calculations – Inhomogeneous computing time with pipelining:



$$\text{Speedup} = \frac{n \sum_{1 \le i \le k} \tau_i}{(k + n - 1) \max(\tau_i)}$$

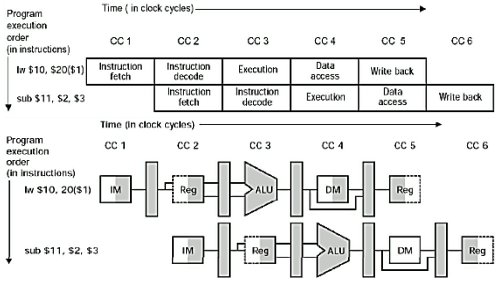
$$\text{Efficiency} = \frac{n}{k \cdot \frac{\max(\tau_i)}{\sum_{1 \le i \le k} \tau_i} \cdot (k + n - 1)} \xrightarrow{n \to \infty} \frac{\sum_{1 \le i \le k} \tau_i}{k \cdot \max_{1 \le i \le k}(\tau_i)}$$

Example on page 9-10

Designing a pipelining architecture:

- start with a single-cycle implementation
- the control path does *not* have an internal state and can therefore be realized as a purely combinational circuit
- *separation* of all stages (IF, ID, EX, MEM, WB) *by registers*
- *assignment of the components* of the data path to the stages
- *exception*: the register field is used in ID and WB (in parallel)

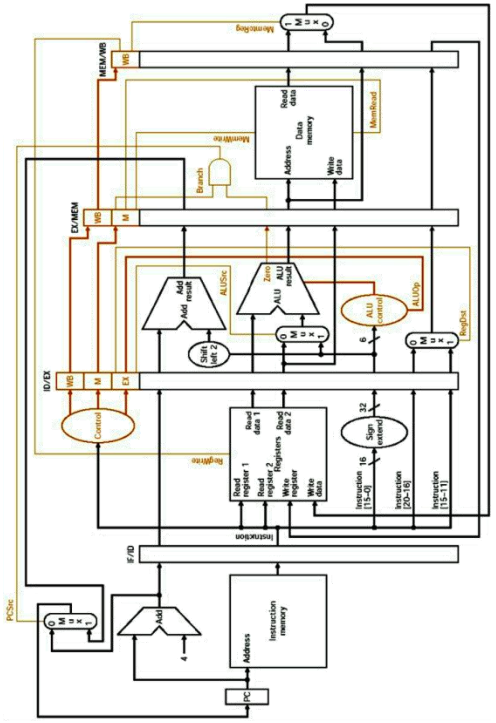
Pipeline Diagram:



Design of the pipelined control path:

- start with the single-cycle implementation of the control path
- lead the control path through the registers (like the data)

Entire Pipelined Implementation without Forwarding:



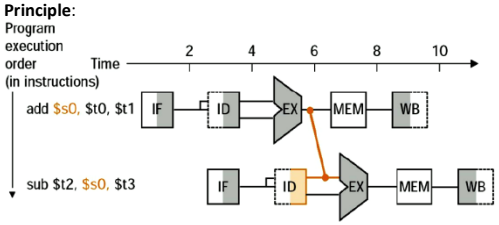
HAZARDS

- ***Structural hazard***: Combination of instructions which should be executed are not supported by the architecture
- ***Flow hazard***: The result of an instruction execution is required to decide which will be the next instruction (branch)
- ***Data hazard***: Operand of an instruction depends on the result of an earlier instruction. (instructions using the same regs)

Preventing Hazards: the ***compiler*** orders the instructions, such that no hazard occurs. If necessary it inserts nop instructions.
Stall: insertion of a bubble into an instruction (nop it & repeat it)

FORWARDING: PREVENT DATA HAZARDS

Principle:



Description of the forwarding functionality:

Notation: `MEM/WB.RegisterRd`
Pipeline Register MEM/WB — Feld mit dem Namen RegisterRd

Weiterleiten eines Datums aus dem EX/MEM-Register, das heisst eines vorherigen ALU-Operanden:

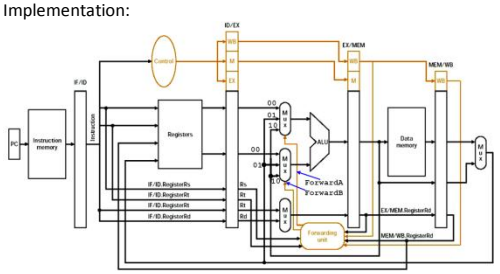
```
if (  EX/MEM.RegWrite = 1      and
      EX/MEM.RegisterRd != 0   and
      EX/MEM.RegisterRd = ID/EX.RegisterRs)
{ForwardA = '10';}
if (  EX/MEM.RegWrite = 1      and
      EX/MEM.RegisterRd != 0   and
      EX/MEM.RegisterRd = ID/EX.RegisterRt)
{ForwardB = '10';}
```

Weiterleiten eines Datums aus dem MEM/WB-Register, das heisst eines vergangenen ALU-Operanden oder eines Speicherzugriffs:

```
if (  MEM/WB.RegWrite = 1      and
      MEM/WB.RegisterRd != 0   and
      EX/MEM.RegisterRd != ID/EX.RegisterRs and
      MEM/WB.RegisterRd = ID/EX.RegisterRs)
{ForwardA = '01';}
if (  MEM/WB.RegWrite = 1      and
      MEM/WB.RegisterRd != 0   and
      EX/MEM.RegisterRd != ID/EX.RegisterRt and
      MEM/WB.RegisterRd = ID/EX.RegisterRt)
{ForwardB = '01';}
```

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

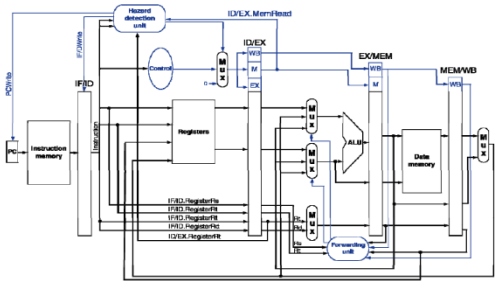
Implementation:



STALLS: PREVENT DATA „LOAD-USE“ HAZARD

- forwarding cannot prevent all data hazards
- insert a ***bubble*** to put things right (like a nop)
- the bubble is not inserted in the IF stage, but later
- all instructions in earlier stages stay there

page 9.39: MEM read and forward -> ??
Implementation:



STALLS: TO PREVENT FLOW HAZARDS

Example, if branching decision is known after MEM-phase: each branch leads to 3 stall cycles.

Verzweigung	IF	ID	EX	MEM	WB			
Instr. i+1/k		IF	0	0		IF	ID	EX
Instr. i+2/k+1							IF	ID
Instr. i+3/k+2								IF

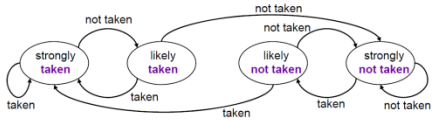
5-stage pipeline, 30% branches → $CPI = 0.3 \cdot 4 + 0.7 = 1.9$

Static Prediction: assume no branching
Example: assuming no branching, but it should have branched

Verzweigung	IF	ID	EX	MEM	WB			
Instr. i+1/k		IF	ID	EX	IF	ID	EX	
Instr. i+2/k+1			IF	ID	O	IF	ID	
Instr. k+2				IF	O	O	IF	

5-stage pipeline, 30% branches, half of the cases prediction right
→ $CPI = 0.5 \cdot 0.3 \cdot 4 + 0.5 \cdot 0.3 + 0.7 = 1.45$

Dynamic 1-bit Prediction: if it brached last time, it will again this time. Useful, because loops usually have the same decision.
Dynamic 2-bit Prediction:



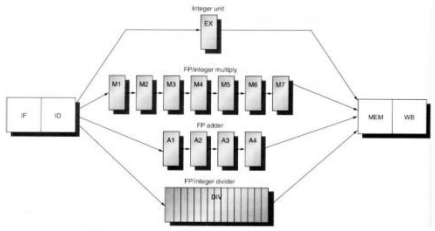
Advance Branching Decision: Already calculate the branch address in the ID-stage
Branch Delay Slot: The instruction following the branch instruction is always executed. So no time is lost.
5-stage pipeline, 30% branches, compiler uses 60% of delay slots
→ $CPI = 0.3 \cdot 2 - 0.3 \cdot 0.6 + 0.7 = 1.12$

INSTRUCTION-LEVEL PARALLELISM (ILP)

Increasing instruction parallelism:

- more pipelinings stages (*superpipelining*)
- multiple instructions per cycle (*multiple issue*): possibility for $CPI < 1$, but dependencies between instructions increase the CPI-value

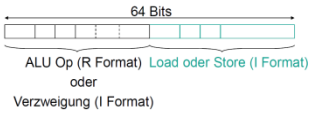
Superpipelining:



consequences: higher frequency, “out of order completion” (different instructions need a different number of cycles), influence of hazards is worse.

MUL/TD	IF	ID	*M1*	M2	M3	M4	M5	M6	**M7**	MEM	WB
ADDD		IF	ID	*A1*	A2	A3	**A4**	MEM	WB		
LD			IF	ID	*EX*	**MEM**	WB				
SD				IF	ID	*EX*	*MEM*	WB			

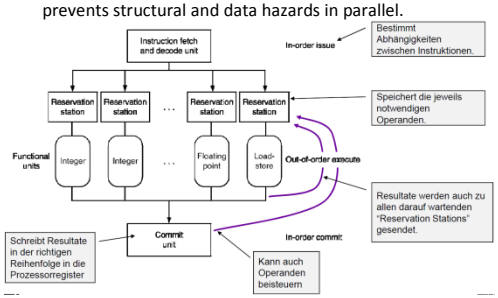
Static Multiple Issue: compiler groups instructions into very long instruction words (VLIW) and prevents hazards. Often arithmetic units are doubled.



Now the registers have to support 4 read and 2 write accesses in parallel and a separate adder for memory addresses is needed

Instruction type	Pipe stages					
ALU or branch instruction	IF	ID	EX	MEM	WB	
Load or store instruction	IF	ID	EX	MEM	WB	
ALU or branch instruction		IF	ID	EX	MEM	WB
Load or store instruction		IF	ID	EX	MEM	WB

Implementation on page 10-9
Compiler Techniques: loop unrolling, examples pages 10-11 ff
Dynamic Multiple Issue: CPU loads multiple instructions and decides which one is run next. Compiler can help prepare this in advance by sorting instructions. CPU solves hazards in real-time using advanced techniques.
Superskalar CPUs: (dynamic multiple issue)
CPU decides on how many instructions are run in parallel and prevents structural and data hazards in parallel.



Register Renaming: Commit unit and reservation stations map logical registers to physical registers. For intermediate results there is no register required sometimes.
Speculation: Estimate what will happen and execute it. Restore original system state if decision was wrong. Is used for dynamic and static multiple issue.
Compiler can reorder instr. and insert other instr. correcting it again, if prediction was wrong (e.g. put an lw before branch)
Hardware can reorder instr. Results are stored until final decision is made. Only then the commit unit really writes it back

STORAGE HIERARCHY (CACHE, ...)

Layers: Register → 1st &2nd Cache → RAM → Hard Drive → Tape
Technology trends: +55% capacity/y, -8% access time/y
Locality:

- time locality: data or instructions which have been used are going to be used again soon → store near CPU

- space locality: data or instructions are used, which are near the just used ones → store near blocks near CPU

Memory Access:
- **Hit**: Data is in the upper layer
- **Miss**: Data needs to be fetched from a lower layer
  → pipeline stall (IF-stage for instr., MEM-stage for data)

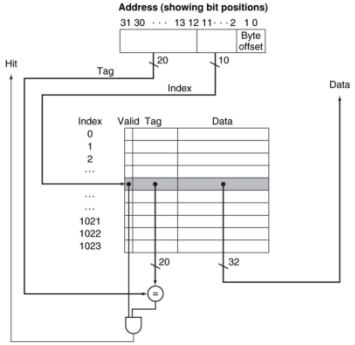
Typical cache values:

Feature	Typical values for L1 caches	Typical values for L2 caches
Total size in blocks	250–2000	15,000–50,000
Total size in kilobytes	16–64	500–4000
Block size in bytes	16–64	64–128
Miss penalty in clocks	10–25	100–1000
Miss rates (global for L2)	2%–5%	0.1%–2%

Terminology:
- *Hit-Rate*: Relativer Anteil der Speicherzugriffe, die in der oberen Ebene erfolgreich sind
- *Hit-Zeit*: Zugriffszeit zur oberen Ebene
  Hit_Zeit = Cache_Zugriffszeit + Zeit_zur_Bestimmung_von_hit_miss
- *Miss-Rate: Relativer Anteil der Speicherzugriffe, die in der oberen Ebene nicht erfolgreich sind*
  Miss_Rate = 1 - (Hit_Rate)
- *Miss-Strafe*:
  Miss_Strafe = Zeit_zum_Finden_eines_Blocks_in_der_unteren_Ebene + Zeit_zum_Übertragen_zur_oberen_Ebene
- *Mittlere Zugriffszeit*:
  Mittlere_Zugriffszeit = Hit_Zeit + Miss_Strafe · Miss_Rate

Associativity – where should a block be placed:
placement of a block with address 12:
direct mapped: 12 mod 8 = 4, set associative: 12 mod 4 = 0, full associative: everywhere possible.

Direct mapped: LSBs of memory address are cache address.



Example access sequence:

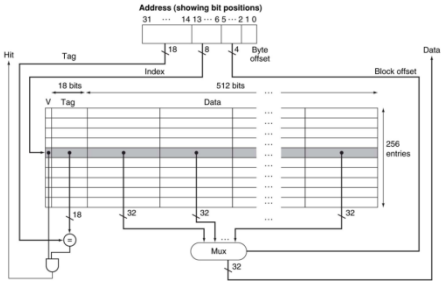
Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss (7.6b)	$(10110_{two} \bmod 8) = 110_{two}$
26	11010_{two}	miss (7.6c)	$(11010_{two} \bmod 8) = 010_{two}$
22	10110_{two}	hit	$(10110_{two} \bmod 8) = 110_{two}$
26	11010_{two}	hit	$(11010_{two} \bmod 8) = 010_{two}$
16	10000_{two}	miss (7.6d)	$(10000_{two} \bmod 8) = 000_{two}$
3	00011_{two}	miss (7.6e)	$(00011_{two} \bmod 8) = 011_{two}$
16	10000_{two}	hit	$(10000_{two} \bmod 8) = 000_{two}$
18	10010_{two}	miss (7.6f)	$(10010_{two} \bmod 8) = 010_{two}$
16	10000_{two}	hit	$(10000_{two} \bmod 8) = 000_{two}$

Further example on page 11-17

Calculating the cache size: 64 kByte data, block size = 1 word = 4byte, byte-wise addressing, address length 32bit
64 kByte = 2^{16} byte = 2^{14} words
cache size = $2^{14}(32 + (32 - 2 - 14) + 1) = 803$ kBit ≈ 100 KB

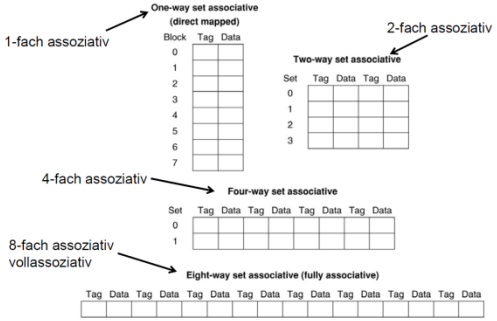
INCREASING THE CACHE BLOCK SIZE

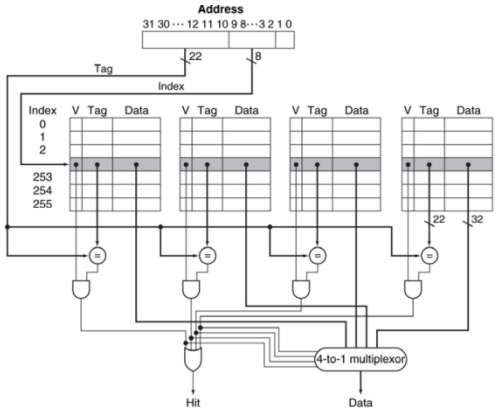
cache block: cache data with their own tag
increasing the block size: profit from space locality, more efficient storage, higher miss rate (≫time for replacement)
memory size: L bit wide address, byte-wise addressing, cache with 2^N bytes usable data, 2^M bytes per block
address parts. $[L-1,\dots,N]$ tag, $[N-1,\dots,M]$ cache index, $[M-1,\dots,0]$ block offset
cache size: $(1 + (L - N) + 8 \cdot 2^M) \cdot 2^{N-M}$ bit
$= 2^N$ byte $+(1 + L - N) \cdot 2^{N-M}$ bit
Generally an optimal block size exists (minimal miss rate)
direct mapped:
Diagramme eines Caches mit direkter Abbildung, 16 kByte Nutzdaten, Blockgrösse 16 Worte, byteweise Adressierung, 32 Bit Adresslänge:



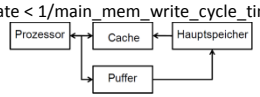
ASSOCIATIVE CACHE

Problems with direct mapping: high miss rate due to a conflict (multiple memory blocks on one cache index), unfortunate replacement strategy, → bigger or associative cache
Associative cache: K entries per index (k-way associative), K direct caches work in parallel, cache index *selects a set of blocks* and compares address in parallel.





Replacement Strategies: For direct mapping we have no choice. For associative cache: random choice or longest unused block.
Write Back Options:
- CPU only writes to cache, if block is replaced, it is copied back
- **dirty bit**: shows whether block was used, copy only if needed
- **write through**: always write back. Needs a buffer. Requires: avg_mem_rate < 1/main_mem_write_cycle_time

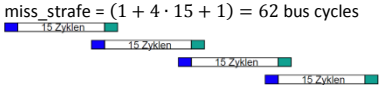


PERFORMANCE CALCULATIONS

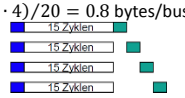
Grundlegende Gleichungen:
CPU_Zeit = (CPU_Instruktionszyklen + CPU_Wartezyklen)·Taktperiode
CPU_Wartezyklen = Lese_Wartezyklen + Schreib_Wartezyklen
Lese_Wartezyklen =
Leseinstruktionen/Programm · Lese_Miss_Rate · Lese_Miss_Strafe/Taktperiode
Schreib_Wartezyklen =
(Schreibinstruktionen/Programm · Schreib_Miss_Rate · Schreib_Miss_Strafe/Taktperiode) + Schreibpuffer_Wartezyklen
Vereinfachte Gleichungen:
CPU_Wartezyklen =
Speicherinstruktionen/Programm · Miss_Rate · Miss_Strafe/Taktperiode = Instruktionen/Programm · Miss/Instruktion · Miss_Strafe/Taktperiode
Example calculations on pages 11-29 ff
Calculations for multiple cache level (L1,L2,L3): pages 11-32 ff.

MEMORY AND BUS

The combination of memory architecture and bus system can influence the overall system performance massively.
Memory organizations: a: one-word-wide, b: wider, c: interleaved (multiple memory banks)
Example: 1 bus cycle to transmit address, 15 bus cycles for each memory access, 1 bus cycle per data transfer
a: block size 4 words, memory & bus width 1 word
miss_strafe = $(1 + 4 \cdot 15 + 1) = 62$ bus cycles

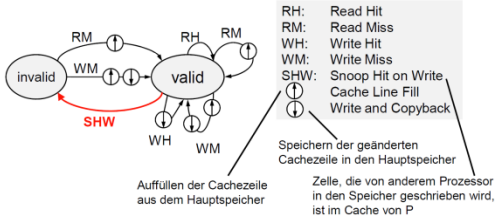


bandwidth = $(4 \cdot 4)/62 = 0.258$ bytes/bus cycle
b: block size 4 words, memory & bus width 4 words
miss_strafe = $(1 + 15 + 1) = 17$ bus cycles
bandwidth = $(4 \cdot 4)/17 = 0.94$ bytes/bus cycle
c: block size 4 words, 4 memory banks, bus width 1 word
miss_strafe = $(1 + 15 + 4) = 20$ bus cycles
bandwidth = $(4 \cdot 4)/20 = 0.8$ bytes/bus cycle



CACHE COHERENCE

Problem of multi processor systems with common memory: incoherent data in caches and main memory.
Example: variable X is in the caches of the CPUs P1 and P2 and in the main mem. P1 writes X=1. With write through the main mem is updated, but P2 read the old values from its cache. Without write through we will get a similar problem.
Snoopy protocols: all CPUs monitor data transmissions between all caches an the main mem. This requires an extension of the status bits of each cache line, an additional cache controller with the according cache coherence protocol. To provide access conflict between CPUs, the address tags and status bits are duplicated (snoop tag).
Protocols are represented by FSMs, if this case states are mapped to cache lines and represent the current situation.
Protocol example: write invalidate for write through, write invalidate for write back, MESI (modified, exclusive, shared invalid)
Write invalidate for write through:
- Das Zustandsdiagramm ist auf eine Cachezeile von Prozessor *P* bezogen.



TODO, …

TODO: add instructions `ll` und `sc`
UNSORTED: `$gp` zeigt immer auf die Mitte der statischen Daten.
MITNEHMEN: Info 1 Zsfg, TIK II Zsg