**MIPS ASSEMBLER**

**REGISTERS**
- pc: Program Counter
- hi: Special Arithmetic Register
- lo: Special Arithmetic Register
- $zero: Constant Value 0

**MIPS SAMPLES**
- lw $t1, 4($t0) #address = 52+4 (Bytes)
- hi-endian: most significant byte at lowest address
- synchronization: li $t0, 0

**SYSCALLS**
- print_int: integer print
- print_float: float print
- print_double: double print
- print_string: string print
- read_int: integer input
- read_float: float input
- read_double: double input
- read_string: string input
- exit: exit main

**ASSEMBLY DIRECTIVES**
- .data [start_addr, end_addr]: Data Segment
- .text [start_addr, end_addr]: Kernel Text Segment
  - Text Segment (actual program)
- .asciiz "str"": String in mem., no null-termination
- .asciii "str"": String in mem., no null-termination

**CORE INSTRUCTION SET**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Format</th>
<th>Comment</th>
<th>Operation (Verilog)</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>R</td>
<td>Add</td>
<td>R(d) = R(n) + R(p)</td>
<td></td>
</tr>
<tr>
<td>addi</td>
<td>R</td>
<td>Add</td>
<td>R(d) = R(n) + Immediate</td>
<td></td>
</tr>
<tr>
<td>addiu</td>
<td>R</td>
<td>Add</td>
<td>R(d) = R(n) + SignExImm</td>
<td></td>
</tr>
<tr>
<td>addu</td>
<td>R</td>
<td>Add</td>
<td>R(d) = R(n) + R(p)</td>
<td></td>
</tr>
<tr>
<td>and</td>
<td>R</td>
<td>And</td>
<td>R(d) = R(n) &amp; R(p)</td>
<td></td>
</tr>
<tr>
<td>andi</td>
<td>R</td>
<td>And</td>
<td>R(d) = R(n) &amp; Immediate</td>
<td></td>
</tr>
<tr>
<td>or</td>
<td>R</td>
<td>Or</td>
<td>R(d) = R(n)</td>
<td>R(p)</td>
</tr>
<tr>
<td>ori</td>
<td>R</td>
<td>Or</td>
<td>R(d) = R(n)</td>
<td>R(p)</td>
</tr>
<tr>
<td>xor</td>
<td>R</td>
<td>Xor</td>
<td>R(d) = R(n) ^ R(p)</td>
<td></td>
</tr>
<tr>
<td>nor</td>
<td>R</td>
<td>Nor</td>
<td>R(d) = ~ (R(n)</td>
<td>R(p)</td>
</tr>
<tr>
<td>srl</td>
<td>R</td>
<td>Srl</td>
<td>R(d) = R(n)&gt;&gt; R(p)</td>
<td></td>
</tr>
<tr>
<td>sra</td>
<td>R</td>
<td>Sra</td>
<td>R(d) = R(n)&gt;&gt; R(p)</td>
<td></td>
</tr>
<tr>
<td>sllv</td>
<td>R</td>
<td>Sllv</td>
<td>R(d) = R(n) &lt;&lt; R(p)</td>
<td></td>
</tr>
<tr>
<td>slti</td>
<td>R</td>
<td>Slti</td>
<td>R(d) = R(n) &lt; R(p)</td>
<td></td>
</tr>
<tr>
<td>sltiu</td>
<td>R</td>
<td>Sltiu</td>
<td>R(d) = R(n) &lt;= R(p)</td>
<td></td>
</tr>
<tr>
<td>sltu</td>
<td>R</td>
<td>Sltu</td>
<td>R(d) = R(n) &lt; R(p)</td>
<td></td>
</tr>
<tr>
<td>sll</td>
<td>R</td>
<td>Sll</td>
<td>R(d) = R(n) &lt;&lt; 1</td>
<td></td>
</tr>
<tr>
<td>sllv</td>
<td>R</td>
<td>Sllv</td>
<td>R(d) = R(n) &lt;&lt; R(p)</td>
<td></td>
</tr>
<tr>
<td>srl</td>
<td>R</td>
<td>Srl</td>
<td>R(d) = R(n)&gt;&gt; R(p)</td>
<td></td>
</tr>
<tr>
<td>sra</td>
<td>R</td>
<td>Sra</td>
<td>R(d) = R(n)&gt;&gt; R(p)</td>
<td></td>
</tr>
<tr>
<td>sllv</td>
<td>R</td>
<td>Sllv</td>
<td>R(d) = R(n) &lt;&lt; R(p)</td>
<td></td>
</tr>
<tr>
<td>slti</td>
<td>R</td>
<td>Slti</td>
<td>R(d) = R(n) &lt; R(p)</td>
<td></td>
</tr>
<tr>
<td>sltiu</td>
<td>R</td>
<td>Sltiu</td>
<td>R(d) = R(n) &lt;= R(p)</td>
<td></td>
</tr>
<tr>
<td>sltu</td>
<td>R</td>
<td>Sltu</td>
<td>R(d) = R(n) &lt; R(p)</td>
<td></td>
</tr>
<tr>
<td>sll</td>
<td>R</td>
<td>Sll</td>
<td>R(d) = R(n) &lt;&lt; 1</td>
<td></td>
</tr>
<tr>
<td>sllv</td>
<td>R</td>
<td>Sllv</td>
<td>R(d) = R(n) &lt;&lt; R(p)</td>
<td></td>
</tr>
<tr>
<td>srl</td>
<td>R</td>
<td>Srl</td>
<td>R(d) = R(n)&gt;&gt; R(p)</td>
<td></td>
</tr>
<tr>
<td>sra</td>
<td>R</td>
<td>Sra</td>
<td>R(d) = R(n)&gt;&gt; R(p)</td>
<td></td>
</tr>
<tr>
<td>sllv</td>
<td>R</td>
<td>Sllv</td>
<td>R(d) = R(n) &lt;&lt; R(p)</td>
<td></td>
</tr>
<tr>
<td>slti</td>
<td>R</td>
<td>Slti</td>
<td>R(d) = R(n) &lt; R(p)</td>
<td></td>
</tr>
<tr>
<td>sltiu</td>
<td>R</td>
<td>Sltiu</td>
<td>R(d) = R(n) &lt;= R(p)</td>
<td></td>
</tr>
<tr>
<td>sltu</td>
<td>R</td>
<td>Sltu</td>
<td>R(d) = R(n) &lt; R(p)</td>
<td></td>
</tr>
</tbody>
</table>

**CORE INSTRUCTION SET PROPERTIES**
- O: May cause overflow exception
- S: ZeroException = (16|Immediate(15)), immediate
- Z: ZeroException = (16|Immediate(15)), immediate
- B: BranchAdd1 = (16|Immediate(15)), immediate, 2’s
- J: JumpAdd1 = (PC|31,28), address, 2’s
- U: Operands considered unsign

**TIPS, TRICKS & COMMON MISTAKES**
- convert for to while and use pointer arithmetics
- after using mu, 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 $t0, ($t1), do not forget brackets and 0
- work from inner to outer loops, assign variables to registers
**INSTRUCTION SET**

Layer Model:
C program \(\rightarrow [\text{Compiler}] \rightarrow [\text{assembly program}] \rightarrow [\text{Assembler}] \rightarrow \) program object code (machine language) \(\rightarrow\) object code from library \(\rightarrow \) Executable \(\rightarrow [\text{Loader}] \rightarrow \) Memory

Neumann-Computing:
Load instructions from memory \(\rightarrow\) decode instructions \(\rightarrow\) fetch operands (from memory or registers) \(\rightarrow\) execute instruction \(\rightarrow\) save result \(\rightarrow\) identify next instruction \(\rightarrow\) [start over]

**ADDRESSING METHODS**

Direct Addressing (also: immediate addressing):
- Register Addressing:
  - 
  - Base Addressing:
    - new PC = constant + old PC + 4

Pseudo-direct Addressing:
- new PC = constant + old PC + 4 + upper 4 bits of old PC + 4

**ASSEMBLER & MACHINE PROGRAM**

Assembler:
- Translates the assembler prog. to a machine prog.
- The assembler program contains: comments, symbolic opcodes, symbolic register names, symbolic marks (lines), macros
- The assembler also handles pseudo instructions and latencies.

Pseudo-Instruction:
- move $00, $11 \(\rightarrow\) add $8, $0, $9
- Latency: load operations (e.g. lw) are only available in the second instruction after 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:
- 
  - function call (callee): Save temporary registers (transfer values from $sa0-$sa3, $st0-$st9, $sv0-$sv1 to stack) \(\rightarrow\) put arguments into $sa0-$sa3 (and on the stack, if needed) \(\rightarrow\) 3: Jump-instruction (jal)

  - (callee): 4: Allocate frame on stack (decrease $sp) \(\rightarrow\) 5: Store saved registers, if needed ($sp, $ra, $sa0-$sa7) \(\rightarrow\) 6: set frame pointer $sp \(\rightarrow\) 7: store results in $sv0, $sv1 \(\rightarrow\) 8: restore saved registers \(\rightarrow\) 9: deallocate frame \(\rightarrow\) 10: jump back (jr $ra)

  - (caller): 11: store temporary registers

**INTERUPTS**

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 routine executed, if an interrupt occurs, corresponding hardware or software mask in $12 is set, global mask set in $12 and the automatic mask is set in $12.
### Performance Assessment

Evaluate a computer: cost, energy consumption, execution time (latency, cpu time), throughput, response time on interrupts (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)

**Typical Memory Layout**

- Stack:undyne data, current program arguments, local variables
- Stack data
- Text: main program, defines the size and address range of text and data
- Initial data: initialization of data
- Program data: copies program text and data into address range. Puts unresolved labels. Includes all the necessary references to libraries.
- Loader: defines the size and address range of text and data segment. Copies program text and data into address range. Puts arguments of the program on the stack, initializes registers.
- Typical memory layout:

```
<table>
<thead>
<tr>
<th>Type</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>undyne data, current</td>
</tr>
<tr>
<td></td>
<td>program arguments, local</td>
</tr>
<tr>
<td></td>
<td>variables</td>
</tr>
<tr>
<td>Stack data</td>
<td></td>
</tr>
<tr>
<td>Text</td>
<td>main program,</td>
</tr>
<tr>
<td></td>
<td>defines the size and</td>
</tr>
<tr>
<td></td>
<td>address range of text and</td>
</tr>
<tr>
<td></td>
<td>data</td>
</tr>
<tr>
<td>Initial</td>
<td>data initialization of</td>
</tr>
<tr>
<td></td>
<td>data</td>
</tr>
<tr>
<td>Program</td>
<td>data</td>
</tr>
<tr>
<td>Data</td>
<td></td>
</tr>
</tbody>
</table>
```

### 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

- New: 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
- Geringen Kosten
- Low Frequency
- High Frequency
- CPI: cycles for instruction type i
- F/2: fraction of instruction i of all instructions of the program
- N/2: # instr. of type i in the program, N: # instr. in the program

### Synchronous Protocols

**Synchronous:** Common clock line for synchronizations, protocol relative to this clock. Pro: fast, if clock skew small.

**Con:** every unit has to support the same frequency

**Example (read from slave):**

```
Address: 0x1234
Data: 0x5678
```

**Timing:**

- `ClkToQ` delay from rising clock edge to valid output register
- `ClkPeriod` delay of the clock signal between registers
- `SetupTime` time the input signal has to be valid before/after rising clock edge
- `SDP` shortest/longest delay between output and input of all registers

### Asynchronous Protocols

**Properties:**

- No clock line, ‘handshake’ protocol
- Pro: common clock, delay independent function, support of heterogeneous units
- Con: requires asynchronous handshaking

**Specification using a Finite State Machine (FSM):**

A deterministic finite state machine is a 6-tupel $\langle I, O, X, y, f, g \rangle$ with:

- $I$: finite input set;
- $O$: finite output set;
- $X$: finite state set

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

```
\begin{align*}
    u & = i \chi & & \text{ if clock skew small} \\
    y & = g(x) & & \text{contextual information} \\
    x & = f(x) & & \text{update function}
\end{align*}
```

### Operating System

The operating system is the interface between the I/O-hardware and the user program 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 programs, mutual exclusion of the users, fair access for all users.

**Ways of communication:**

- Instructions (OS → I/O): Addressing: memory mapped I/O (like memory address, but instead of memory a device)
- Messages (I/O → OS): Polling: periodic sampling of the status registers
- Interrupt: Unit interrupts current program (special hardware), usage of cause/status-registers
- Data (OS → I/O): 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, …

**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
  - sub
  - addi
  - and
  - or
  - oriu
  - ori
  - sll
  - srl

- **Speichersstruktionen:**
  - load
  - store

- **Verzweigungssstruktionen:**
  - j
  - jal

**Elementary Operations:**

- Der Prozessor stellt die folgenden Elementaroperationen und Variablen zur Verfügung:
  - **Zustandsvariablen:** zum Speichern von 32 Bit Daten und Instruktionen: pc: ALU, pc: PC (program counter), spc: stack pointer, spc: spc (command counter), spc: spc (instruction)
  - **Interne Register:**
    - reg: 32, 1-31
    - reg: 32, 64, 32-63
    - reg: 32, 32-63
  - **Hauptspeicher an der Adresse:**
  - **Verwendung eines Wortes:**
  - **Aneinanderhängen:**
  - **Ruckwärtsanordnung:**
  - **Arithmetische Operation:**

**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 output 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

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

Unified net (R, lw, sw, but not beq)

**Control Path:** Represents the structure of the Petri net:

**ALU:** a component of the data path

**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:**

- **single cycle**
  - **Control Path**
    - ALU Control & Control Units
    - 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
  - **Calculations**
    - **Homogeneous computing time with pipelining:**
      - Speedup = \( \frac{n}{k} \), Efficiency = \( \frac{k}{k - n - 1} \)
      - 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
- the register field is used in ID and WB (in parallel)

**Pipeline Diagram:**

**Calculations**

- **Inhomogeneous computing time with pipelining:**
  - Speedup = \( \frac{n}{k} \), Efficiency = \( \frac{k - n - 1}{k} \)
  - Example on page 9-10

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:

Description of the forwarding functionality:
- Notation: MEM/Reg[register]
- Pipeline Register MEM/Reg[register]
- Forwarding of a datum from the EX/MEM-Register, which helps avoid hazards.

Entirely of a Datum aus dem EX/MEM-Register, das (see example)
- lead of a hazard avoid ALU-Operanden oder eines
- Spaltenzugriffs.

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.

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

Example, if branching decision is known after MEM-phase: each branch leads to 3 stall cycles.
5-stage pipeline, 70% branches $\rightarrow$ CPI = 0.3 · 4 + 0.7 = 1.9
Static Prediction: assume no branching
Example: assuming no branching, but it should have branched

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

Implementation on page 10-9

Compiler Techniques:
- loop unrolling, examples pages 10-11
- 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.
- Superscalar CPUs: (dynamic multiple issue)
- CPU decides on how many instructions are run in parallel and prevents structural and data hazards in parallel.

STORAGE HIERARCHY (CACHE, ...)
Terminology:
- Hit-Rate: Relative Anteil der Speicherzugriffe, die in der obener Ebene stattfinden
- Hit-Zeil: Zugriff auf die obere Ebene
- Miss-Zeil: Zugriff auf die untere Ebene
- Miss-Rate: Relative Anteil der Speicherzugriffe, die in der unteren Ebene stattfinden
- Miss-Strate: Miss-Tage / Summe der fehlerhaften Adressen
- Miss Rate: 1 - Miss Rate

Direct mapped: 12 mod 8 = 4, set associative: 12 mod 4 = 0, full associative: everywhere possible.

Example access sequence:
```
Address [1][2][3][4][5][6][7][8][9][10][11][12]
```

Further example on page 11
Calculating the cache size: 64 kByte data, block size = 1 word = 8 byte, byte-wise addressing, address length 32bit
64 kByte = 2^26 byte = 2^32 words
Cache size = 2^10 * (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 Access:
- Miss: Data needs to be fetched from a lower layer
- Hit: Data is in the upper layer

Example access sequence:
```
Example access sequence:
```

Terminology:
- Miss Rate: Relative Anteil der Speicherzugriffe, die in der obener Ebene stattfinden
- Hit Rate: Zugriff auf die obere Ebene
- Miss Rate: Relative Anteil der Speicherzugriffe, die in der unteren Ebene stattfinden
- Hit Rate: 1 - Miss Rate

Direct mapped: 12 mod 8 = 4, set associative: 12 mod 4 = 0, full associative: everywhere possible.

Example access sequence:
```
Address [1][2][3][4][5][6][7][8][9][10][11][12]
```

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.

The combination of memory architecture and bus system can influence the overall system performance massively.

Memory organizations:
- a: oneway-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 stride = (1 + 4 + 15 + 1) = 62 bytes cycles
```

b: block size 4 words, memory & bus width 4 words
```
miss stride = (1 + 15 + 1) = 17 bus cycles
```

Memory and Bus

Calculations for multiple cache level (L1, L2, L3): pages 11-17

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 on 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 Zustandsempfinden ist auf eine Cachezeile von Prozessor P bezogen.

Write invalidate:

INFO: 1 Zfgd, TIK II Zsfg