# T01 Single-Cycle Processors

School of Electrical and Computer Engineering  
Cornell University

revision: 2014-09-03-17-21

## 1 Instruction Set Architecture  
### 1.1 IBM 360 Instruction Set Architecture  
### 1.2 MIPS32 Instruction Set Architecture  
### 1.3 PARC Instruction Set Architecture

## 2 Single-Cycle Processors  
### 2.1 Single-Cycle Processor Datapath  
### 2.2 Single-Cycle Processor Control Unit

## 3 Analyzing Processor Performance
1. Instruction Set Architecture

• By early 1960’s, IBM had several incompatible lines of computers!
  – Defense : 701
  – Scientific : 704, 709, 7090, 7094
  – Business : 702, 705, 7080
  – Mid-Sized Business : 1400
  – Decimal Architectures : 7070, 7072, 7074

• Each system had its own:
  – Implementation and potentially even technology
  – Instruction set
  – I/O system and secondary storage (tapes, drums, disks)
  – Assemblers, compilers, libraries, etc
  – Application niche

• IBM 360 was the first line of machines to separate ISA from microarchitecture
  – Enabled same software to run on different current and future microarchitectures
  – Reduced impact of modifying the microarchitecture enabling rapid innovation in hardware

... the structure of a computer that a machine language programmer must understand to write a correct (timing independent) program for that machine.

— Amdahl, Blaauw, Brooks, 1964
ISA is the contract between software and hardware

1. 
   - Representations for characters, integers, floating-point
   - Integer formats can be signed or unsigned
   - Floating-point formats can be single- or double-precision
   - Byte addresses can be ordered within a word as either little- or big-endian

2. 
   - Registers: general-purpose, floating-point, control
   - Memory: different addresses spaces for heap, stack, I/O

3. 
   - Register: operand stored in registers
   - Immediate: operand is part of instruction
   - Direct: address of operand in memory is stored in instruction
   - Register Indirect: address of operand in memory is stored in register
   - Displacement: register indirect, addr is added to immediate
   - Autoincrement/decrement: register indirect, addr is automatically adj
   - PC-Relative: displacement is added to the program counter

4. 
   - Integer and floating-point arithmetic instructions
   - Register and memory data movement instructions
   - Control transfer instructions
   - System control instructions

5. 
   - Opcode, addresses of operands and destination, next instruction
   - Variable length vs. fixed length
1.1. IBM 360 Instruction Set Architecture

• How is data represented?
  – 8-bit bytes, 16-bit half-words, 32-bit words, 64-bit double-words
  – IBM 360 is why bytes are 8-bits long today!

• Where can data be stored?
  – $2^{24}$ 32-bit memory locations
  – 16 general-purpose 32-bit registers and 4 floating-point 64-bit registers
  – Condition codes, control flags, program counter

• What operations can be done on data?
  – Large number of arithmetic, data movement, and control instructions
1. Instruction Set Architecture

1.1. IBM 360 Instruction Set Architecture

<table>
<thead>
<tr>
<th></th>
<th>Model 30</th>
<th>Model 70</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage</td>
<td>8–64 KB</td>
<td>256–512 KB</td>
</tr>
<tr>
<td>Datapath</td>
<td>8-bit</td>
<td>64-bit</td>
</tr>
<tr>
<td>Circuit Delay</td>
<td>30 ns/level</td>
<td>5 ns/level</td>
</tr>
<tr>
<td>Local Store</td>
<td>Main store</td>
<td>Transistor registers</td>
</tr>
<tr>
<td>Control Store</td>
<td>Read only 1µs</td>
<td>Conventional circuits</td>
</tr>
</tbody>
</table>

- IBM 360 instruction set architecture completely hid the underlying technological differences between various models

- **Significant Milestone**: The first true ISA designed as a portable hardware-software interface

- **IBM 360**: 45 years later ...
  The zSeries Z196 Microprocessor
  - 5.2 GHz in IBM 45 nm SOI
  - 1.4B transistors in 512 mm²
  - Four cores per chip
  - Aggressive out-of-order execution
  - Four-level cache hierarchy with embedded DRAM L3/L4
  - Can still run IBM 360 code!

— Hot Chips 2010
1.2. MIPS32 Instruction Set Architecture

• How is data represented?
  – 8-bit bytes, 16-bit half-words, 32-bit words
  – 32-bit single-precision, 64-bit double-precision floating point

• Where can data be stored?
  – $2^{32}$ 32-bit memory locations
  – 32 general-purpose 32-bit registers, 32 SP (16 DP) floating-point registers
  – FP status register, Program counter

• How can data be accessed?
  – Register, register indirect, displacement

• What operations can be done on data?
  – Large number of arithmetic, data movement, and control instructions

• How are instructions encoded?
  – Fixed-length 32-bit instructions

MIPS R2K: 1986, single-issue, in-order, off-chip caches, 2 µm, 8–15 MHz, 110K transistors, 80 mm²

MIPS R10K: 1996, quad-issue, out-of-order, on-chip caches, 0.35 µm, 200 MHz, 6.8M transistors, 300 mm²
Add Immediate Unsigned Word

Format:  ADDIU rt, rs, immediate

Purpose: Add Immediate Unsigned Word

To add a constant to a 32-bit integer

Description: GPR[rt] ← GPR[rs] + immediate

The 16-bit signed immediate is added to the 32-bit value in GPR rs and the 32-bit arithmetic result is placed into GPR rt.

No Integer Overflow exception occurs under any circumstances.

Restrictions:

None

Operation:

temp ← GPR[rs] + sign_extend(immediate)
GPR[rt] ← temp

Exceptions:

None

Programming Notes:

The term “unsigned” in the instruction name is a misnomer; this operation is 32-bit modulo arithmetic that does not trap on overflow. This instruction is appropriate for unsigned arithmetic, such as address arithmetic, or integer arithmetic environments that ignore overflow, such as C language arithmetic.
Load Word

Format: \texttt{LW rt, offset(base)}

Purpose: Load Word

To load a word from memory as a signed value

Description: \texttt{GPR[rt] \leftarrow memory[GPR[base] + offset]}

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched, sign-extended to the GPR register length if necessary, and placed in GPR \texttt{rt}. The 16-bit signed \texttt{offset} is added to the contents of GPR \texttt{base} to form the effective address.

Restrictions:

The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

Operation:

\begin{verbatim}
vAddr \leftarrow \text{sign}\_\text{extend}(\text{offset}) + \text{GPR}\[\text{base}]
\text{if} vAddr_{1..0} \neq 0^\circ\text{ then}
\quad \text{SignalException(AddressError)}
\text{endif}
(pAddr, CCA) \leftarrow \text{AddressTranslation}(vAddr, \text{DATA}, \text{LOAD})
\text{memword} \leftarrow \text{LoadMemory}(CCA, \text{WORD}, pAddr, vAddr, \text{DATA})
\text{GPR}\[rt] \leftarrow \text{memword}
\end{verbatim}

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error, Watch
Branch on Not Equal

Format:  BNE rs, rt, offset

Purpose:  Branch on Not Equal
To compare GPRs then do a PC-relative conditional branch

Description:  if GPR[rs] ≠ GPR[rt] then branch
An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.
If the contents of GPR rs and GPR rt are not equal, branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:
Processor operation is UNPREDICTABLE if a branch, jump, ERET, DERET, or WAIT instruction is placed in the delay slot of a branch or jump.

Operation:
I:  target_offset ← sign_extend(offset || 0^2)
    condition ← (GPR[rs] ≠ GPR[rt])
I+1:  if condition then
         PC ← PC + target_offset
       endif

Exceptions:
None

Programming Notes:
With the 18-bit signed instruction offset, the conditional branch range is ±128 KBytes. Use jump (J) or jump register (JR) instructions to branch to addresses outside this range.
1.3. PARC Instruction Set Architecture

• Subset of MIPS32 with several important differences
  – Only little-endian, very simple address translation
  – No hi/lo registers, only 32 general purpose registers
  – Multiply and divide instructions target general purpose registers
  – Only a subset of all MIPS32 instructions
  – No branch delay slot

• PARCv1: Very small subset suitable for examples

• PARCv2: Subset suitable for executing simple C programs without system calls (i.e., open, write, read)

• PARCv3: Subset suitable for executing all integer single-threaded C programs without system calls

• PARCv4: Full PARC ISA
PARCv1 instruction assembly, semantics, and encoding

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Assembly</th>
<th>Semantics</th>
<th>Encoding</th>
<th>PC Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>addiu rt, rs, imm</td>
<td>( R[rt] \leftarrow R[rs] + \text{sext(imm)} )</td>
<td>( R[rt] \leftarrow R[rs] + \text{sext(imm)} )</td>
<td>001001 rs rt imm</td>
<td>31 26 25 21 20 16 15 11 10 0</td>
</tr>
<tr>
<td>lw rt, imm(rs)</td>
<td>( R[rt] \leftarrow M[ R[rs] + \text{sext(imm)} ] )</td>
<td>( R[rt] \leftarrow M[ R[rs] + \text{sext(imm)} ] )</td>
<td>100011 rs rt imm</td>
<td>31 26 25 21 20 16 15 11 10 0</td>
</tr>
<tr>
<td>sw rt, imm(rs)</td>
<td>( M[ R[rs] + \text{sext(imm)} ] \leftarrow R[rt] )</td>
<td>( M[ R[rs] + \text{sext(imm)} ] \leftarrow R[rt] )</td>
<td>101011 rs rt imm</td>
<td>31 26 25 21 20 16 15 11 10 0</td>
</tr>
<tr>
<td>j target</td>
<td>PC ← ( j\text{targ}( PC, \text{target} ) )</td>
<td>( PC \leftarrow j\text{targ}( PC, \text{target} ) )</td>
<td>000010 target</td>
<td>31 26 25 10 0</td>
</tr>
<tr>
<td>jal target</td>
<td>R[31] ← PC + 4</td>
<td>( R[31] \leftarrow PC + 4 )</td>
<td>000011 target</td>
<td>31 26 25 10 0</td>
</tr>
<tr>
<td>jr rs</td>
<td>( PC \leftarrow R[rs] )</td>
<td>( PC \leftarrow R[rs] )</td>
<td>000000 rs 00000 00000 00000 001000</td>
<td>31 26 25 21 20 16 15 11 10 6 5 0</td>
</tr>
<tr>
<td>bne rs, rt, imm</td>
<td>if (R[rs] ≠ R[rt])</td>
<td>( PC \leftarrow PC + 4 + \text{imm} \times 4 )</td>
<td>000101 rs 0 imm</td>
<td>31 26 25 21 20 16 15 11 10 6 5 0</td>
</tr>
</tbody>
</table>
PARCv1 vector-vector add assembly and C program

C code for doing element-wise vector addition.

Equivalent PARCv1 assembly code. Recall that arguments are passed in registers r4–r7, return value is stored to r2, and the return address is stored in r31.
PARCv1 mystery assembly and C program

What is the C code corresponding to the PARCv1 assembly shown below? Assume assembly implements a function.

```
addiu r12, r0, 0

loop:
  lw  r13, 0(r4)
  bne r13, r6, foo
  addiu r2, r12, 0
  jr  r31

foo:
  addiu r4, r4, 4
  addiu r12, r12, 1
  bne  r12, r5, loop

  addiu r2, r0, -1
  jr   r31
```
2. Single-Cycle Processors

Control Unit

Control Signals

Status Signals

Datapath

Datapath

<1 cycle combinational

Memory

Inst Val

Inst Req

Inst Resp

Data Val

Data Req

Data Resp

<1 cycle combinational
2.1. Single-Cycle Processor Datapath

Implementing ADDU Instruction

Implementing ADDIU Instruction
Implementing ADDU and ADDIU Instruction

Adding the MUL Instruction
Adding the LW Instruction

Adding the SW Instruction
Adding the J Instruction

Adding the JAL Instruction
Adding the JR Instruction

Adding the BNE Instruction
Adding a New Auto-Incrementing Load Instruction

Draw on the datapath diagram what paths we need to use as well as any new paths we will need to add in order to implement the following auto-incrementing load instruction.

\[
\text{lw.ai } rt, \text{ imm}(rs) \\
R[rt] \leftarrow M[R[rs] + \text{sext}(\text{imm})]; R[rs] \leftarrow R[rs] + 4
\]
2.2. Single-Cycle Processor Control Unit

<table>
<thead>
<tr>
<th>inst</th>
<th>pc_sel</th>
<th>op1_sel</th>
<th>alu_func</th>
<th>wb_sel</th>
<th>waddr</th>
<th>wen</th>
</tr>
</thead>
<tbody>
<tr>
<td>addu</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addiu</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mul</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jal</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw.ai</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3. **Analyzing Processor Performance**

\[
\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycles}}
\]

- Instructions / program depends on source code, compiler, ISA
- Cycles / instruction (CPI) depends on ISA, microarchitecture
- Time / cycle depends upon microarchitecture and implementation

<table>
<thead>
<tr>
<th>Microarchitecture</th>
<th>CPI</th>
<th>Cycle Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>this topic (\rightarrow)</td>
<td>1</td>
<td>long</td>
</tr>
<tr>
<td>Topic 02 FSM Processor</td>
<td>&gt;1</td>
<td>short</td>
</tr>
<tr>
<td>Topic 03 Pipelined Processor</td>
<td>(\approx 1)</td>
<td>short</td>
</tr>
</tbody>
</table>

Students often confuse “Cycle Time” with the execution time of a sequence of transactions measured in cycles. “Cycle Time” is the clock period or the inverse of the clock frequency.
Estimating cycle time

There are many paths through the design that start at a state element and end at a state element. The “critical path” is the longest path across all of these paths. We can usually use a simple first-order static timing estimate to estimate the cycle time (i.e., the clock period and thus also the clock frequency).
Estimating execution time

Using our first-order equation for processor performance, how long in nanoseconds will it take to execute the vector-vector add example assuming \( n \) is 64?

```assembly
loop:
  lw    r12, 0(r4)
  lw    r13, 0(r5)
  addu  r14, r12, r13
  sw    r14, 0(r6)
  addiu r4, r4, 4
  addiu r5, r5, 4
  addiu r6, r6, 4
  addiu r7, r7, -1
  bne   r7, r0, loop
  jr     r31
```

Using our first-order equation for processor performance, how long in nanoseconds will it take to execute the mystery program assuming \( n \) is 64 and that we find a match on the 10th element.

```assembly
    addiu r12, r0, 0
loop:
  lw    r13, 0(r4)
  bne   r13, r6, foo
  addiu r2, r12, 0
  jr     r31
foo:
  addiu r4, r4, 4
  addiu r12, r12, 1
  bne   r12, r5, loop
  addiu r2, r0, -1
  jr     r31
```