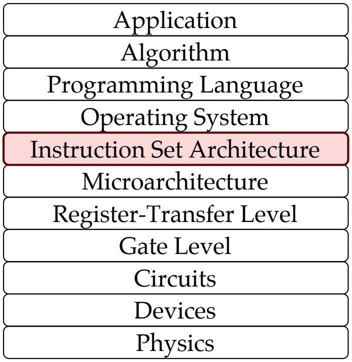
ECE 4750 Computer Architecture, Fall 2015
T01 Fundamental Processor Concepts

School of Electrical and Computer Engineering
Cornell University

revision: 2015-09-05-13-33

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.

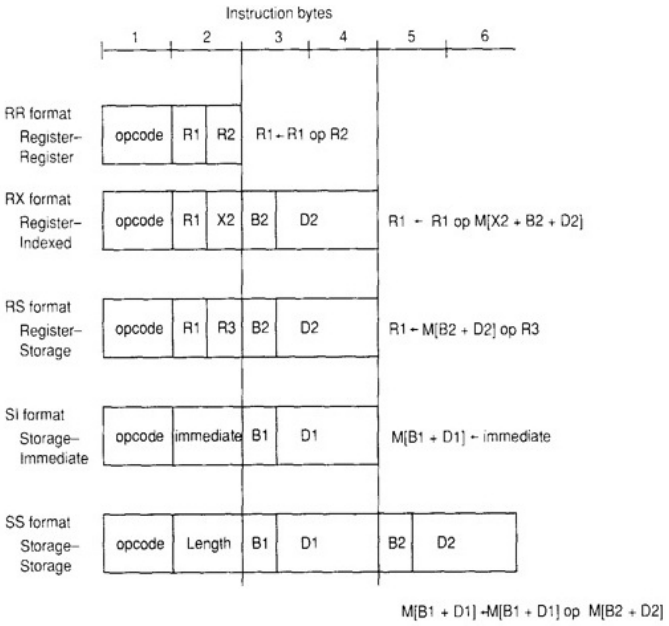
— Amdahl, Blaauw, Brooks, 1964

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 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 an immediate in the 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

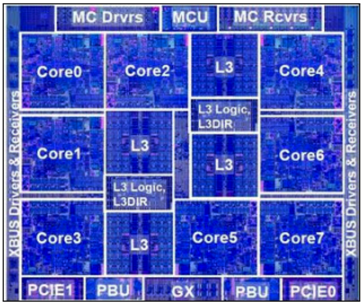
- How is data represented?
  - 8-bit bytes, 16-bit half-words, 32-bit words, 64-bit double-words
  - IBM 360 is why bytes are 8-bits long today!
- Where can data be stored?
  - 2^{24} 32-bit memory locations
  - 16 general-purpose 32-bit registers and 4 floating-point 64-bit registers
  - Condition codes, control flags, program counter
- What operations can be done on data?
  - Large number of arithmetic, data movement, and control instructions



Instruction bytes
1
2
3
4
5
6
RR format Register–Register
opcode
R1
R2
R1 ← R1 op R2
RX format Register–Indexed
opcode
R1
X2
B2
D2
R1 ← R1 op M[X2 + B2 + D2]
RS format Register–Storage
opcode
R1
R3
B2
D2
R1 ← M[B2 + D2] op R3
SI format Storage–Immediate
opcode
immediate
B1
D1
M[B1 + D1] ← immediate
SS format Storage–Storage
opcode
Length
B1
D1
B2
D2
M[B1 + D1] ← M[B1 + D1] op M[B2 + D2]

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

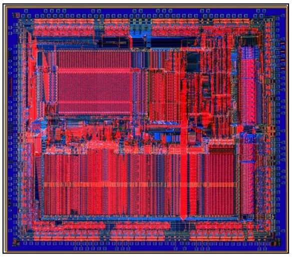
- IBM 360 instruction set architecture completely hid the underlying technological differences between various models
- Significant Milestone: The first true ISA designed as a portable hardware-software interface
- IBM 360: 50 years later ...
  The zSeries z13 Microprocessor
  - 5 GHz in IBM 22 nm SOI
  - 4B transistors in 678 mm^2
  - 17 metal layers
  - ≈20K pads
  - Eight cores per chip
  - Aggressive out-of-order execution
  - Four-level cache hierarchy
  - On-chip 64MB eDRAM L3 cache
  - Off-chip 480MB eDRAM L4 cache
  - Can still run IBM 360 code!



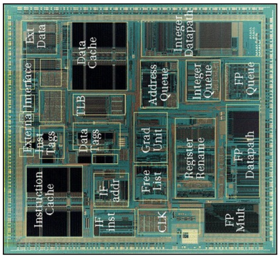
J. Warnock, et al., “22nm Next-Generation IBM System-Z Microprocessor,” Int’l Solid-State Circuits Conference, Feb. 2015.

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 mm^2



MIPS R10K: 1996, quad-issue, out-of-order, on-chip caches, 0.35 μm, 200 MHz, 6.8M transistors, 300 mm^2

31 26	25 21	20 16	15 0
ADDIU 001001	rs	rt	immediate
6	5	5	16

Format: `ADDIU rt, rs, immediate` **MIPS32**

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.

31 26	25 21	20 16	15 0
LW 100011	base	rt	offset
6	5	5	16

Format: `LW rt, offset(base)` **MIPS32**

Purpose: Load Word

To load a word from memory as a signed value

Description: `GPR[rt] ← 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 *rt*. The 16-bit signed *offset* is added to the contents of GPR *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:

```
vAddr ← sign_extend(offset) + GPR[base]
if $vAddr_{1..0} \neq 0^2$ then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
GPR[rt] ← memword
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error, Watch

31 26	25 21	20 16	15 0
BNE 000101	rs	rt	offset
6	5	5	16

Format: `BNE rs, rt, offset` **MIPS32**

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

`http://www.csl.cornell.edu/courses/ece4750/handouts`

- 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`)
  - `subu, and, or, nor, xor, andi, ori, xori, lui`
  - `slt, sltu, slti, sltiu, sll, srl, sra, srav, srlv, sllv`
  - `bgtz, bltz, bgez, blez`
  - `mfc0, mtc0 (stats_en, core_id, num_cores)`
- **PARCv3:** Subset suitable for executing real C/C++ single-threaded and parallel programs with system calls
  - `jalr`
  - `div, divu, rem, remu`
  - `lb, lbu, lh, lhu, sb, sh`
  - `movn, movz`
  - `amo.add, amo.and, amo.or, sync`
  - `syscall, eret`
  - `add.s, sub.s, mul.s, div.s, c.<cond>.s, cvt.s.w, trunc.w.s`
  - `mtx, mfx, mtxr, mfxr`

PARCv1 instruction assembly, semantics, and encoding

`addu rd, rs, rt`
$\mathrm{R}[\texttt{rd}] \leftarrow \mathrm{R}[\texttt{rs}] + \mathrm{R}[\texttt{rt}]$
$\mathrm{PC} \leftarrow \mathrm{PC}{+}4$

6	5	5	5	5	6
000000	rs	rt	rd	00000	100001
31 26	25 21	20 16	15 11	10 6	5 0

`addiu rt, rs, imm`
$\mathrm{R}[\texttt{rt}] \leftarrow \mathrm{R}[\texttt{rs}] + \mathrm{sext}(\texttt{imm})$
$\mathrm{PC} \leftarrow \mathrm{PC}{+}4$

6	5	5	16
001001	rs	rt	offset
31 26	25 21	20 16	15 0

`mul rd, rs, rt`
$\mathrm{R}[\texttt{rd}] \leftarrow \mathrm{R}[\texttt{rs}] \times \mathrm{R}[\texttt{rt}]$
$\mathrm{PC} \leftarrow \mathrm{PC}{+}4$

6	5	5	5	5	6
011100	rs	rt	rd	00000	000010
31 26	25 21	20 16	15 11	10 6	5 0

`lw rt, offset(rs)`
$\mathrm{R}[\texttt{rt}] \leftarrow \mathrm{M}[\ \mathrm{R}[\texttt{rs}] + \mathrm{sext}(\texttt{offset})\ ]$
$\mathrm{PC} \leftarrow \mathrm{PC}{+}4$

6	5	5	16
100011	rs	rt	offset
31 26	25 21	20 16	15 0

`sw rt, offset(rs)`
$\mathrm{M}[\ \mathrm{R}[\texttt{rs}] + \mathrm{sext}(\texttt{offset})\ ] \leftarrow \mathrm{R}[\texttt{rt}]$
$\mathrm{PC} \leftarrow \mathrm{PC}{+}4$

6	5	5	16
101011	rs	rt	offset
31 26	25 21	20 16	15 0

`j targ`
$\mathrm{PC} \leftarrow \{\ (\mathrm{PC}+4)[31{:}28], \texttt{targ}, 00\ \}$

6	26
000010	targ
31 26	25 0

`jal targ`
$\mathrm{R}[31] \leftarrow \mathrm{PC} + 4;$
$\mathrm{PC} \leftarrow \{\ (\mathrm{PC}+4)[31{:}28], \texttt{targ}, 00\ \}$

6	26
000011	targ
31 26	25 0

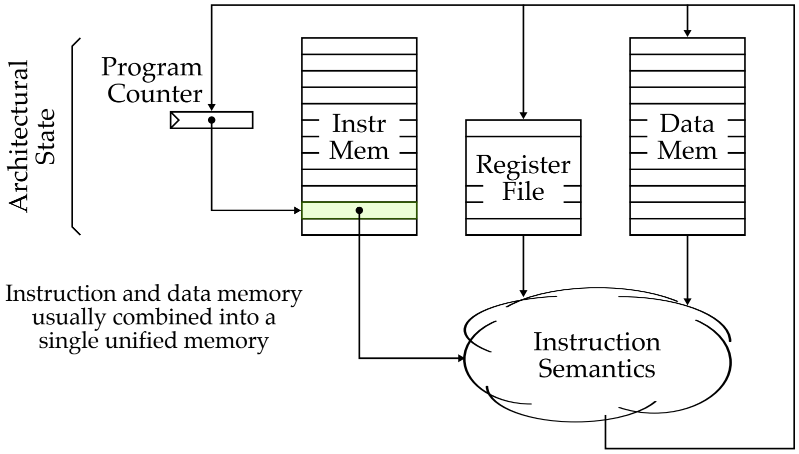
`jr rs`
$\mathrm{PC} \leftarrow \mathrm{R}[\texttt{rs}]$

6	5	5	5	5	6
000000	rs	00000	00000	00000	001000
31 26	25 21	20 16	15 11	10 6	5 0

`bne rs, rt, offset`
$\mathrm{if}\ (\ \mathrm{R}[\texttt{rs}]\ != \mathrm{R}[\texttt{rt}]\ )$
$\quad \mathrm{PC} \leftarrow (\ \mathrm{PC} + 4 + (4 \times \mathrm{sext}(\texttt{offset}))$

6	5	5	16
000101	rs	rt	offset
31 26	25 21	20 16	15 0

2. Processor Functional-Level Model



2.1. Transactions and Steps

- We can think of each instruction as a transaction
- Executing a transaction involves a sequence of steps

	addu	addiu	mul	lw	sw	j	jal	jr	bne
Fetch Instruction									
Decode Instruction									
Read Registers									
Register Arithmetic									
Read Memory									
Write Memory									
Write Registers									
Update PC									

2.2. Simple Assembly Example

Static Asm Sequence	Instruction Semantics
`loop: lw    r1, 0(r2)`	
`      addu  r3, r3, r1`	
`      addiu r2, r2, 4`	
`      bne   r1, r0, loop`	

Worksheet illustrating processor functional-level model

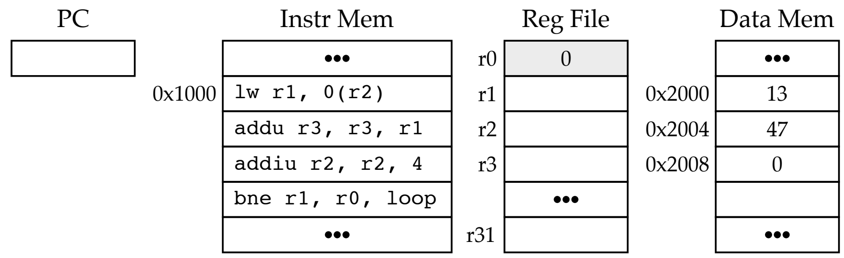


Table illustrating processor functional-level model

PC	Dynamic Asm Sequence	r1	r2	r3
	`lw    r1, 0(r2)`			
	`addu  r3, r3, r1`			
	`addiu r2, r2, 4`			
	`bne   r1, r0, loop`			
	`lw    r1, 0(r2)`			
	`addu  r3, r3, r1`			

2.3. 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 `r4`–`r7`, return value is stored to `r2`, and return address is stored in `r31`.

Note that we are ignoring the fact that our assembly code will not function correctly if `n` <= 0. Our assembly code would need an additional check before entering the loop to ensure that `n` > 0. Unless otherwise stated, we will assume in this course that array bounds are greater than zero to simplify our analysis.

2.4. PARCv1 Mystery Assembly and C Program

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

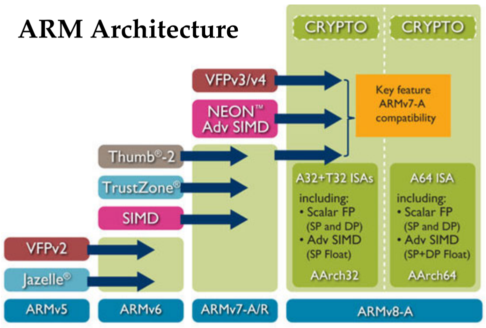
```
  addiu r12, r0, 0

 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
```

3. Processor/Laundry Analogy

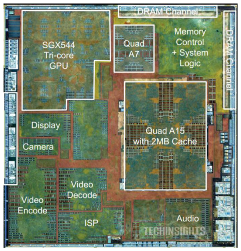
- Processor
  - Instructions are "transactions" that execute on a processor
  - Architecture: defines the hardware/software interface
  - Microarchitecture: how hardware executes sequence of instructions
- Laundry
  - Cleaning a load of laundry is a "transaction"
  - Architecture: high-level specification, dirty clothes in, clean clothes out
  - Microarchitecture: how laundry room actually processes multiple loads

3.1. Arch vs. µArch vs. VLSI Impl

ARM Architecture

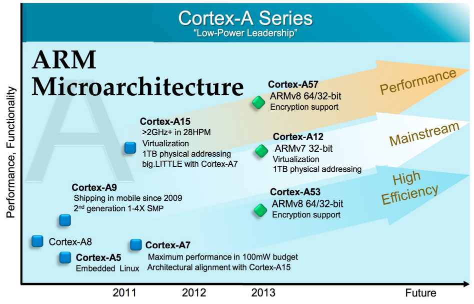

CRYPTO
CRYPTO
VFPv3/v4
NEON™ Adv SIMD
Key feature ARMv7-A compatibility
Thumb®-2
TrustZone®
SIMD
VFPv2
Jazelle®
A32+T32 ISAs including: • Scalar FP (SP and DP) • Adv SIMD (SP Float)
AArch32
A64 ISA including: • Scalar FP (SP and DP) • Adv SIMD (SP+DP Float)
AArch64
ARMv5
ARMv6
ARMv7-A/R
ARMv8-A

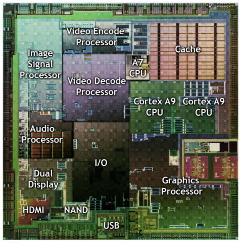
ARM VLSI Implementation



Samsung Exynos Octa

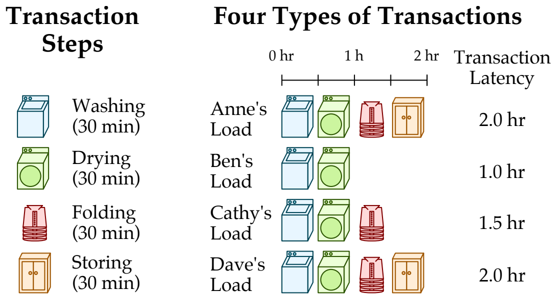
ARM Microarchitecture





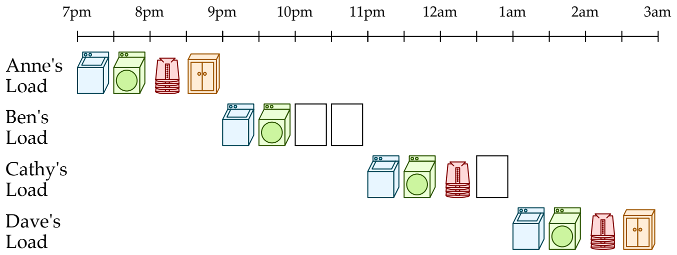
NVIDIA Tegra 2

3.2. Processor Microarchitectural Design Patterns

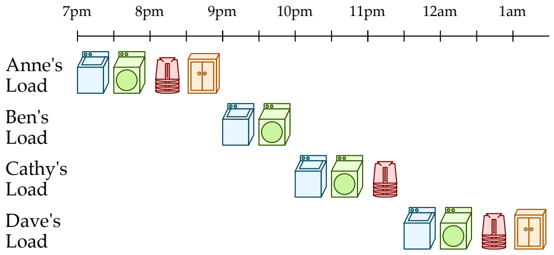


	Transaction Latency	
Anne's Load	2.0 hr	Anne requires all four steps
Ben's Load	1.0 hr	Ben is messy, leaves unfolded clothes in his laundry basket
Cathy's Load	1.5 hr	Cathy does not have a bureau, leaves folded clothes in basket
Dave's Load	2.0 hr	Dave requires all four steps

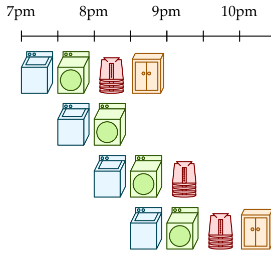
Fixed Time Slot Laundry (Single-Cycle Processors)



Variable Time Slot Laundry (FSM Processors)

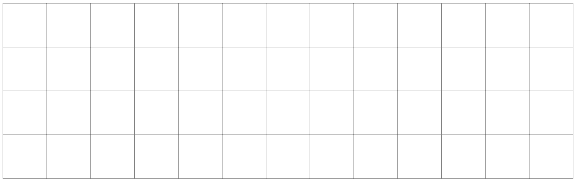


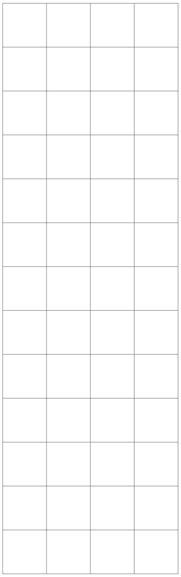
Pipelined Laundry



3.3. Transaction Diagrams

W: Washing **D**: Drying **F**: Folding **S**: Storing





Key Concepts

- Transaction latency is the time to complete a single transaction
- Execution time or total latency is the time to complete a sequence of transactions
- Throughput is the number of transactions executed per unit time

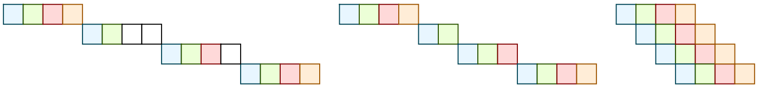
4. Analyzing Processor Performance

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

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

Using our first-order equation for processor performance and a functional-level model, the execution time is just the number of dynamic instructions.

Microarchitecture	**CPI**	**Cycle Time**
Single-Cycle Processor	1	long
FSM Processor	>1	short
Pipelined Processor	≈ 1	short



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 dynamic instruction count

Estimate the dynamic instruction count for the vector-vector add example assuming `n` is 64?

```
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
```

Estimate the dynamic instruction count for the mystery program assuming `n` is 64 and that we find a match on the final element.

```
 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
```