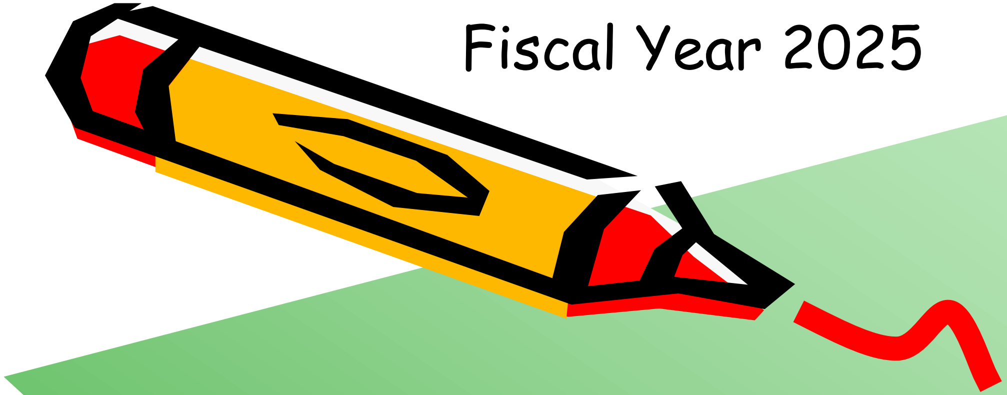


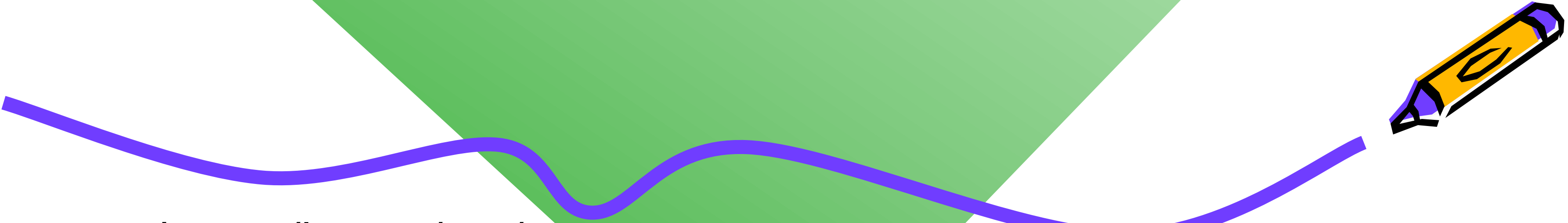
Fiscal Year 2025

Ver. 2025-12-03a

Course number: CSC.T440
School of Computing,
Graduate major in Computer Science

Computer Organization and Architecture

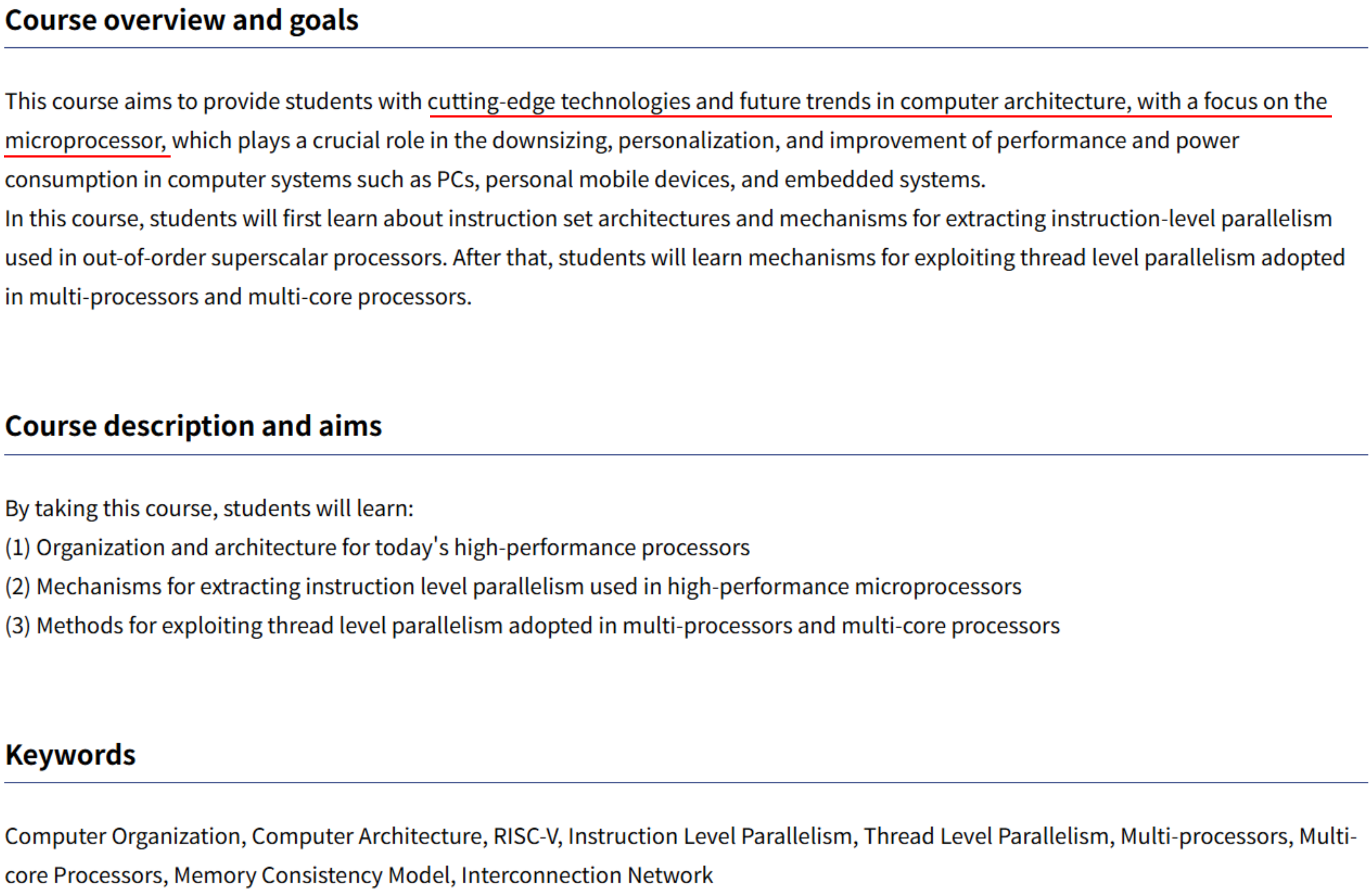
1. Computer Organization and Architecture



www.arch.cs.titech.ac.jp/lecture/coa/
Room No. M-112(H117), Lecture (Face-to-face)
Thr 13:30-15:10

Kenji Kise, Department of Computer Science
kise[at]comp.isct.ac.jp

Syllabus (1/4)



Course overview and goals

This course aims to provide students with cutting-edge technologies and future trends in computer architecture, with a focus on the microprocessor, which plays a crucial role in the downsizing, personalization, and improvement of performance and power consumption in computer systems such as PCs, personal mobile devices, and embedded systems.
In this course, students will first learn about instruction set architectures and mechanisms for extracting instruction-level parallelism used in out-of-order superscalar processors. After that, students will learn mechanisms for exploiting thread level parallelism adopted in multi-processors and multi-core processors.

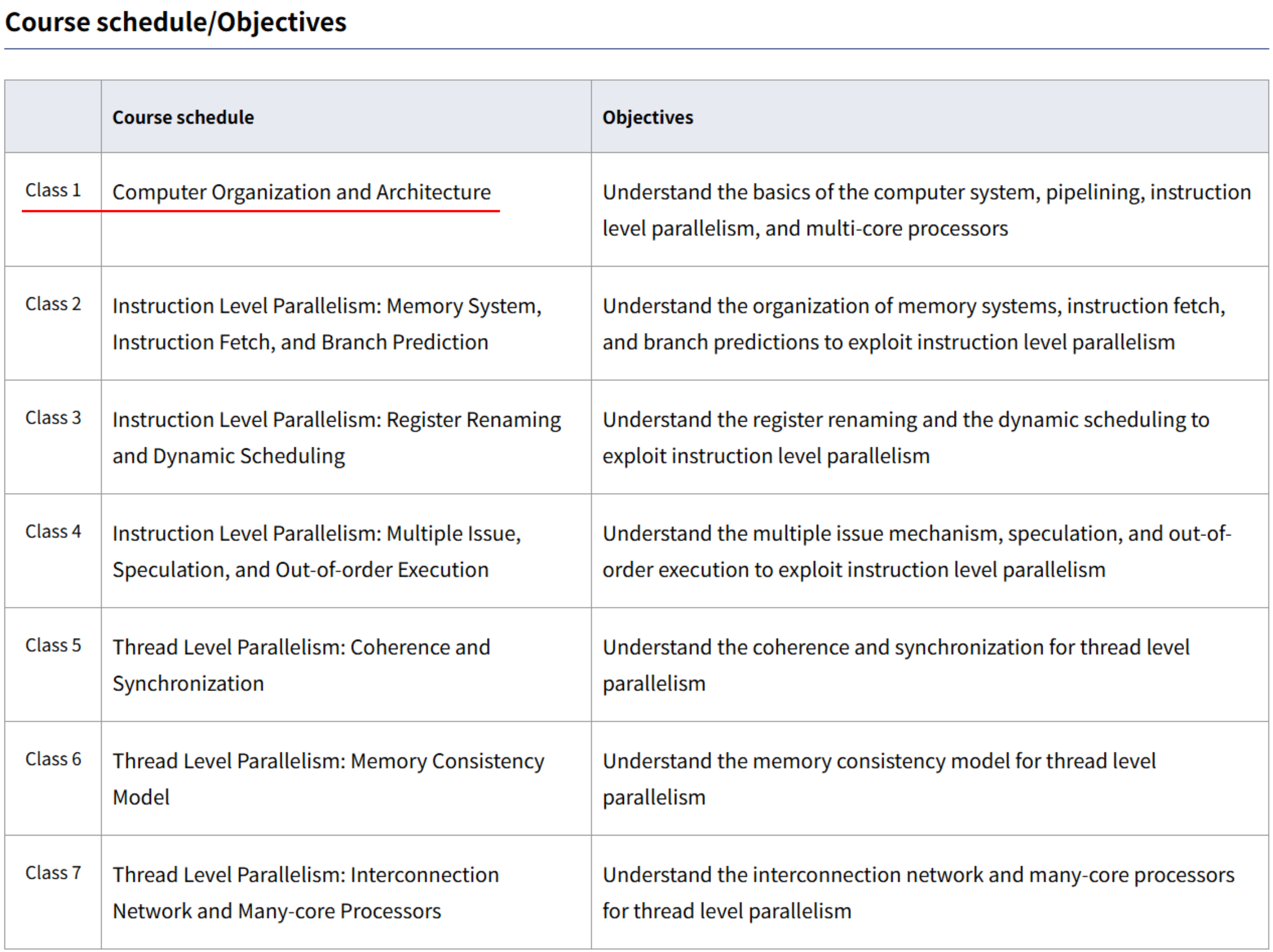
Course description and aims

By taking this course, students will learn:
(1) Organization and architecture for today's high-performance processors
(2) Mechanisms for extracting instruction level parallelism used in high-performance microprocessors
(3) Methods for exploiting thread level parallelism adopted in multi-processors and multi-core processors

Keywords

Computer Organization, Computer Architecture, RISC-V, Instruction Level Parallelism, Thread Level Parallelism, Multi-processors, Multi-core Processors, Memory Consistency Model, Interconnection Network

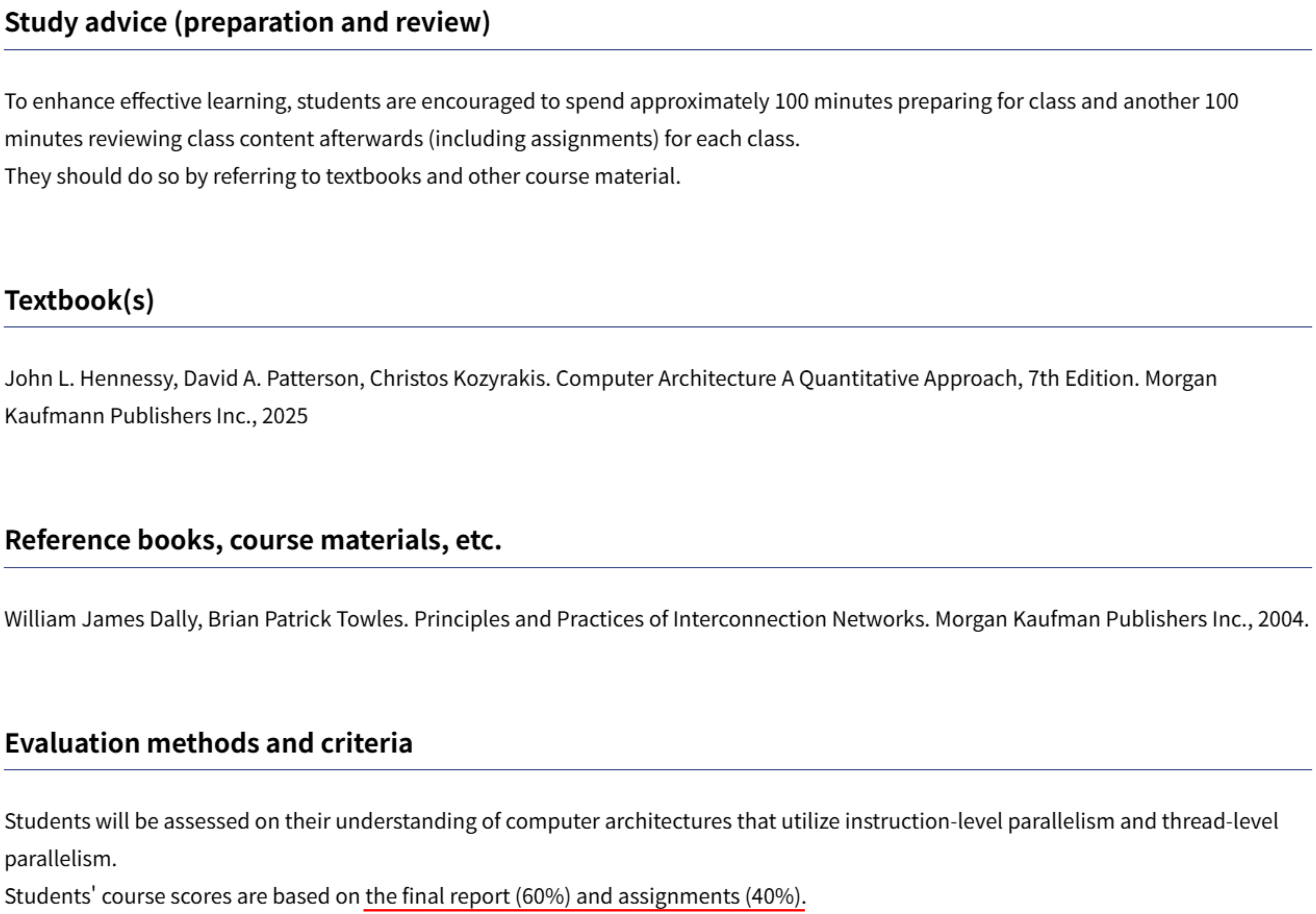
Syllabus (2/4)



Course schedule/Objectives

	Course schedule	Objectives
Class 1	Computer Organization and Architecture	Understand the basics of the computer system, pipelining, instruction level parallelism, and multi-core processors
Class 2	Instruction Level Parallelism: Memory System, Instruction Fetch, and Branch Prediction	Understand the organization of memory systems, instruction fetch, and branch predictions to exploit instruction level parallelism
Class 3	Instruction Level Parallelism: Register Renaming and Dynamic Scheduling	Understand the register renaming and the dynamic scheduling to exploit instruction level parallelism
Class 4	Instruction Level Parallelism: Multiple Issue, Speculation, and Out-of-order Execution	Understand the multiple issue mechanism, speculation, and out-of-order execution to exploit instruction level parallelism
Class 5	Thread Level Parallelism: Coherence and Synchronization	Understand the coherence and synchronization for thread level parallelism
Class 6	Thread Level Parallelism: Memory Consistency Model	Understand the memory consistency model for thread level parallelism
Class 7	Thread Level Parallelism: Interconnection Network and Many-core Processors	Understand the interconnection network and many-core processors for thread level parallelism

Syllabus (3/4)



Study advice (preparation and review)

To enhance effective learning, students are encouraged to spend approximately 100 minutes preparing for class and another 100 minutes reviewing class content afterwards (including assignments) for each class.
They should do so by referring to textbooks and other course material.

Textbook(s)

John L. Hennessy, David A. Patterson, Christos Kozyrakis. Computer Architecture A Quantitative Approach, 7th Edition. Morgan Kaufmann Publishers Inc., 2025

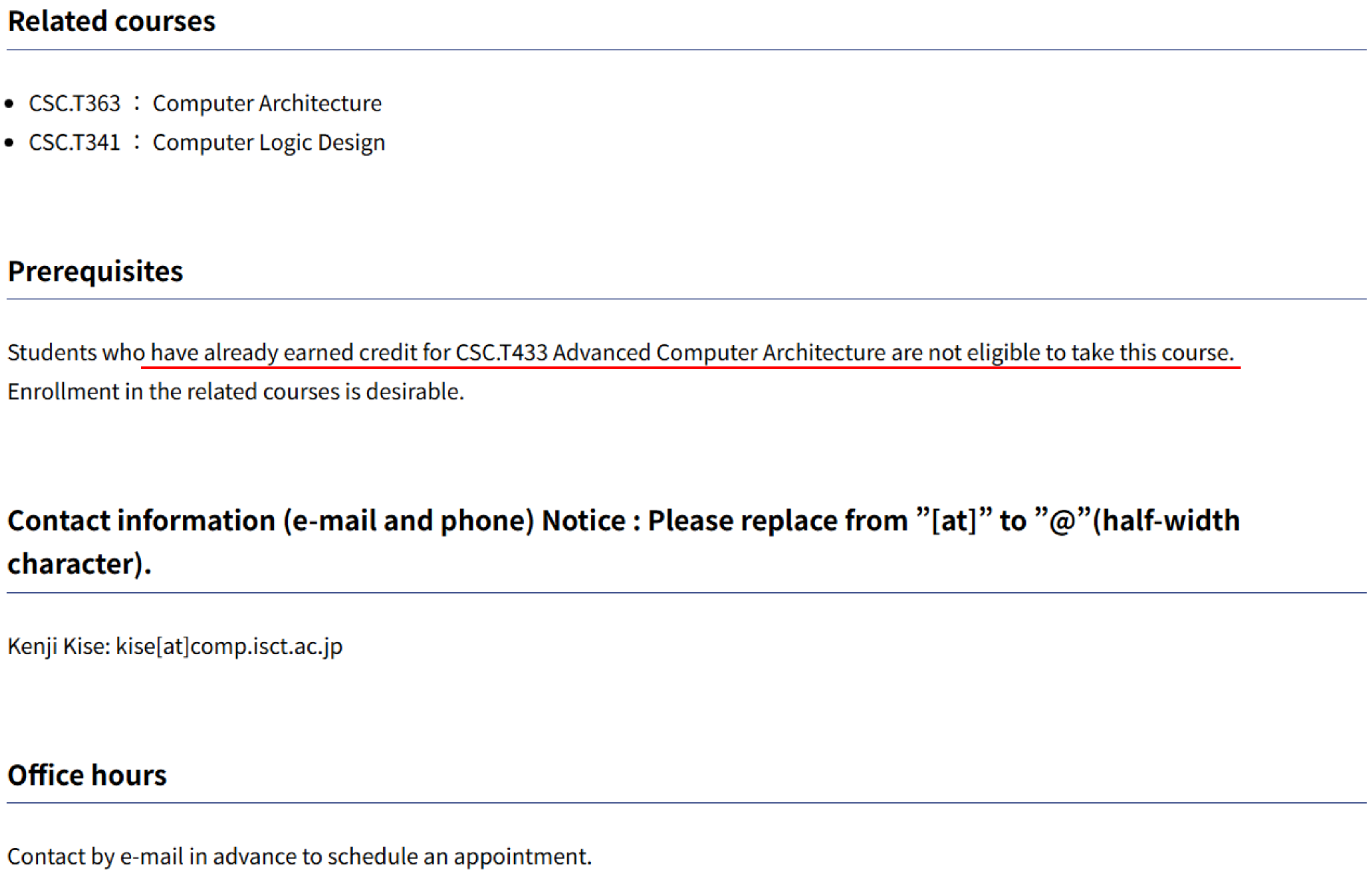
Reference books, course materials, etc.

William James Dally, Brian Patrick Towles. Principles and Practices of Interconnection Networks. Morgan Kaufman Publishers Inc., 2004.

Evaluation methods and criteria

Students will be assessed on their understanding of computer architectures that utilize instruction-level parallelism and thread-level parallelism.
Students' course scores are based on the final report (60%) and assignments (40%).

Syllabus (4/4)



Related courses

- CSC.T363 ： Computer Architecture
- CSC.T341 ： Computer Logic Design

Prerequisites

Students who have already earned credit for CSC.T433 Advanced Computer Architecture are not eligible to take this course.
Enrollment in the related courses is desirable.

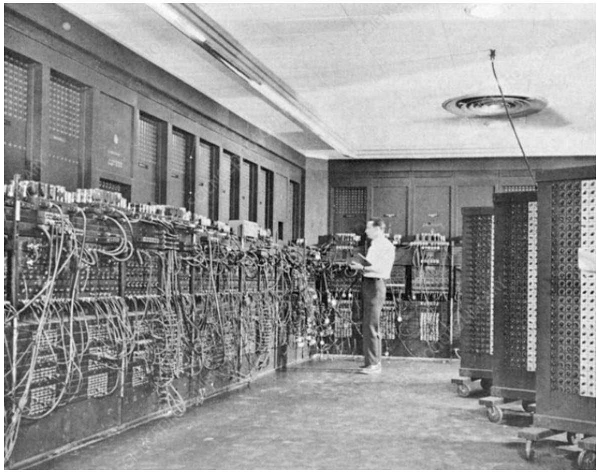
Contact information (e-mail and phone) Notice : Please replace from ”[at]” to ”@”(half-width character).

Kenji Kise: kise[at]comp.isct.ac.jp

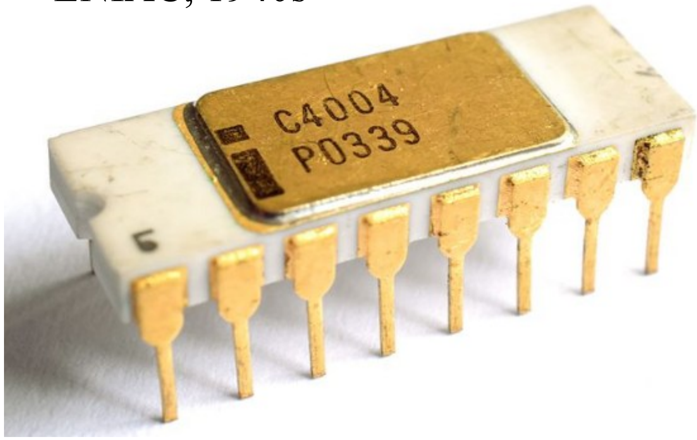
Office hours

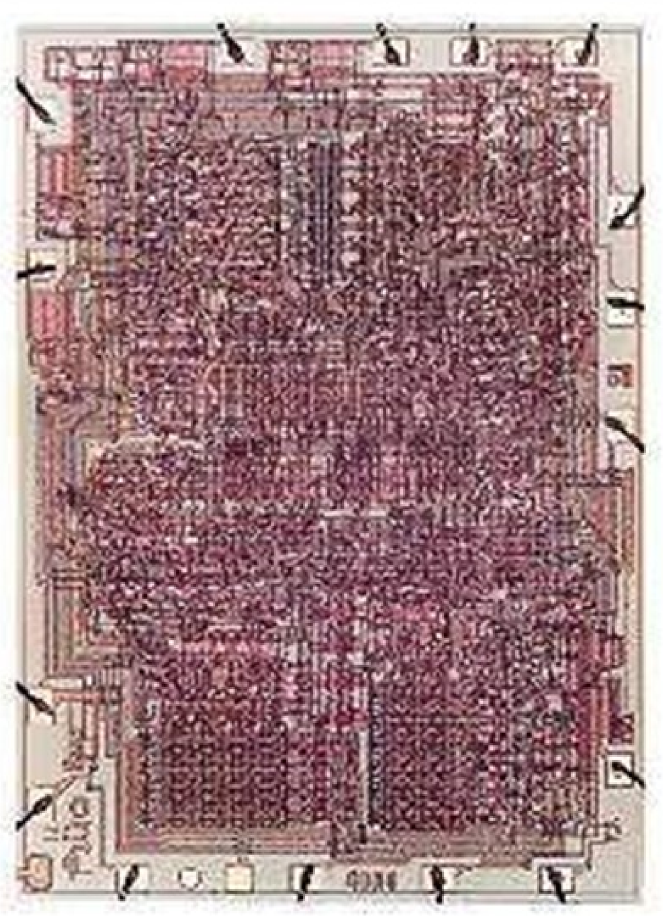
Contact by e-mail in advance to schedule an appointment.

The birth of microprocessors in 1971



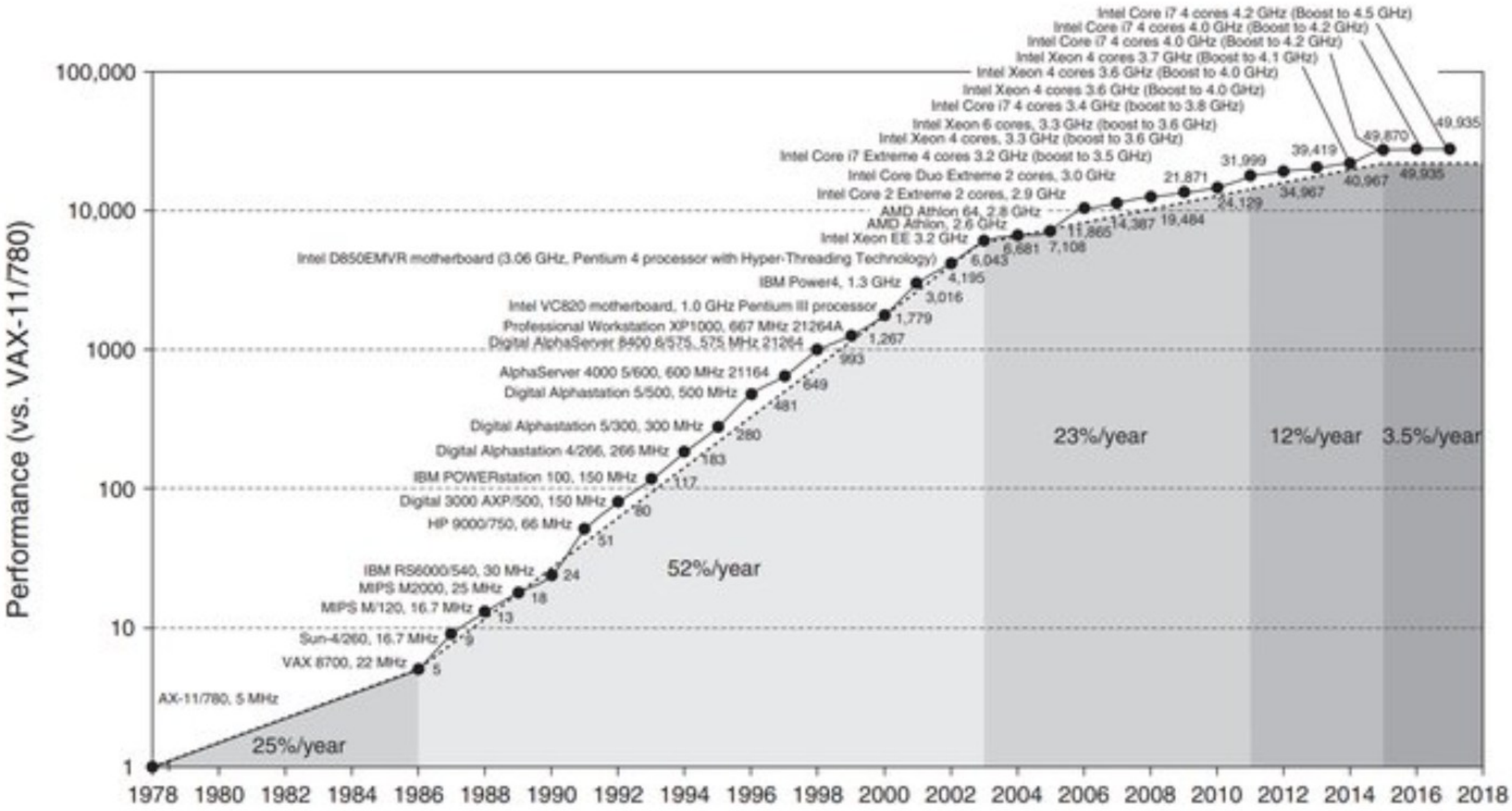
ENIAC, 1940s





Name	Year	# of transistors
Intel 4004	1971	2,250

Growth in processor performance



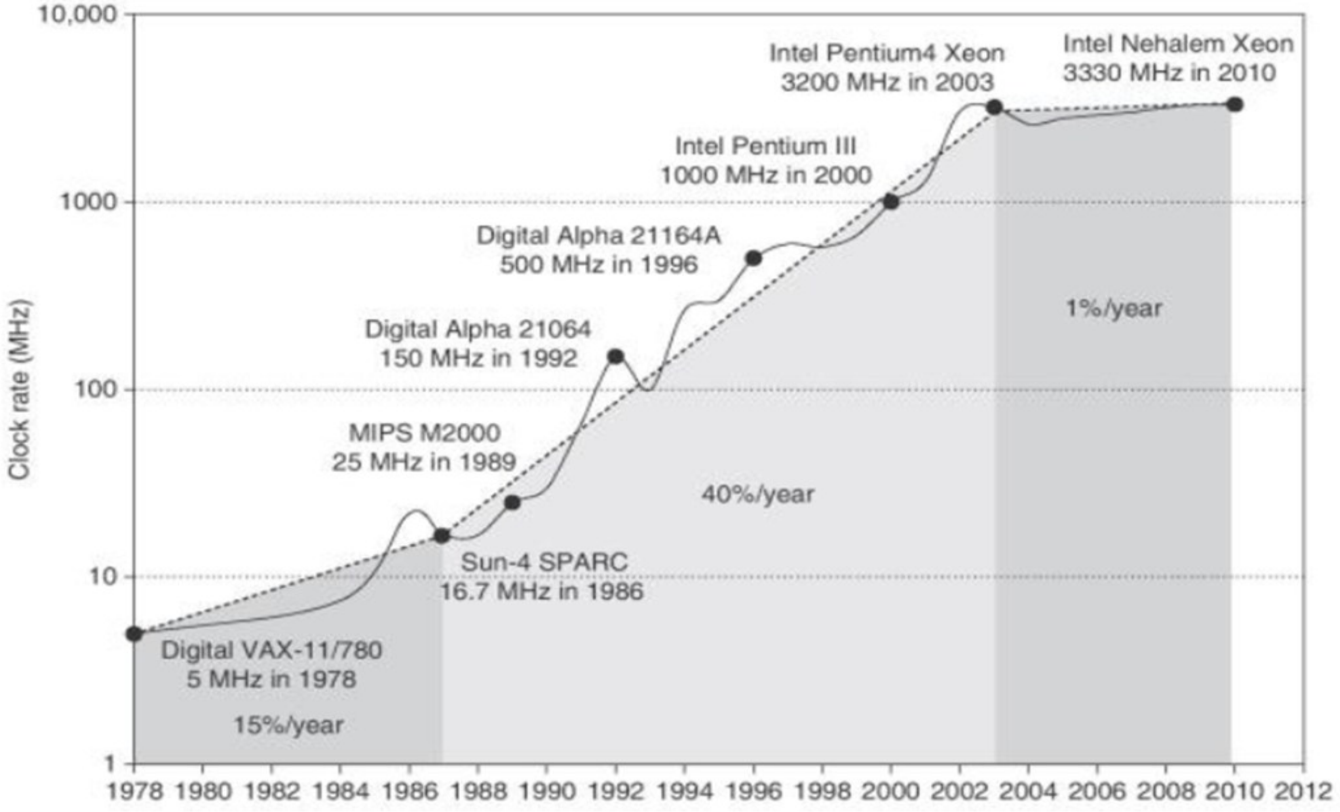
Performance Factors

- Performance = F x IPC
  - F : frequency (clock rate)
  - IPC : executed instructions per cycle (due to architecture advances)

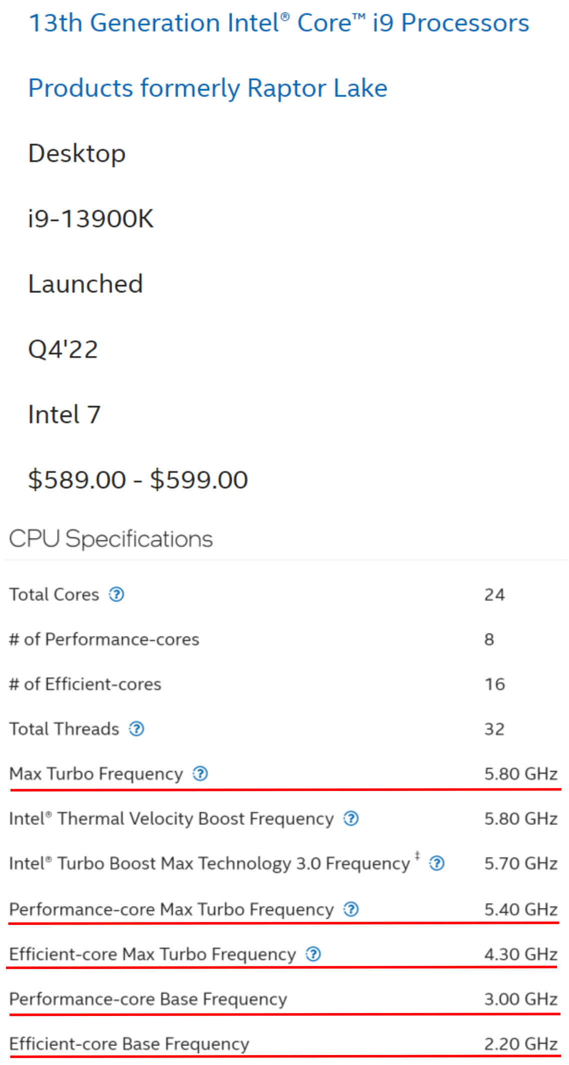
- The performance can be improved by increasing either F or IPC

Growth in clock rate F of microprocessors

Intel 4004 clocked at 740KHz in 1971



From CAQA 5th edition



13th Generation Intel® Core™ i9 Processors

Products formerly Raptor Lake

Desktop

i9-13900K

Launched

Q4'22

Intel 7

$589.00 - $599.00

CPU Specifications

Total Cores ⓘ	24
# of Performance-cores	8
# of Efficient-cores	16
Total Threads ⓘ	32
Max Turbo Frequency ⓘ	5.80 GHz
Intel® Thermal Velocity Boost Frequency ⓘ	5.80 GHz
Intel® Turbo Boost Max Technology 3.0 Frequency ‡ ⓘ	5.70 GHz
Performance-core Max Turbo Frequency ⓘ	5.40 GHz
Efficient-core Max Turbo Frequency ⓘ	4.30 GHz
Performance-core Base Frequency	3.00 GHz
Efficient-core Base Frequency	2.20 GHz

The past, present, and future of the world's most important device

The Best of Hands On › The aspirational delight of our top DIY projects P.18

Stronger Software, Fewer Bugs › The case for functional programming P.40

The Next Step in Worker Productivity › EEG Brain Scans P.46

FOR THE TECHNOLOGY INSIDER

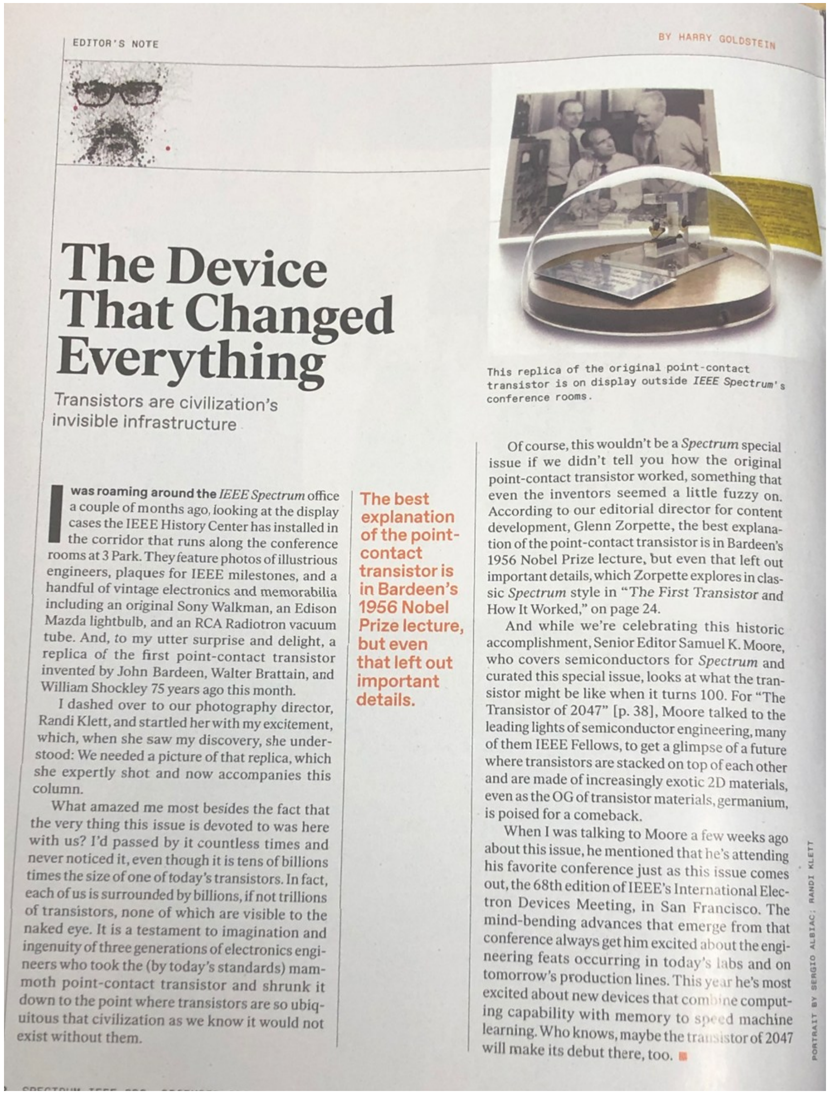
DECEMBER 2022

IEEE Spectrum

The Transistor at 75

The Past, Present, and Future of the World's Most Important Device

EEE



EDITOR'S NOTE

BY HARRY GOLDSTEIN

The Device That Changed Everything

Transistors are civilization's invisible infrastructure

This replica of the original point-contact transistor is on display outside *IEEE Spectrum*'s conference rooms.

I was roaming around the *IEEE Spectrum* office a couple of months ago, looking at the display cases the IEEE History Center has installed in the corridor that runs along the conference rooms at 3 Park. They feature photos of illustrious engineers, plaques for IEEE milestones, and a handful of vintage electronics and memorabilia including an original Sony Walkman, an Edison Mazda lightbulb, and an RCA Radiotron vacuum tube. And, to my utter surprise and delight, a replica of the first point-contact transistor invented by John Bardeen, Walter Brattain, and William Shockley 75 years ago this month.

I dashed over to our photography director, Randi Klett, and startled her with my excitement, which, when she saw my discovery, she understood: We needed a picture of that replica, which she expertly shot and now accompanies this column.

What amazed me most besides the fact that the very thing this issue is devoted to was here with us? I'd passed by it countless times and never noticed it, even though it is tens of billions times the size of one of today's transistors. In fact, each of us is surrounded by billions, if not trillions of transistors, none of which are visible to the naked eye. It is a testament to imagination and ingenuity of three generations of electronics engineers who took the (by today's standards) mammoth point-contact transistor and shrunk it down to the point where transistors are so ubiquitous that civilization as we know it would not exist without them.

The best explanation of the point-contact transistor is in Bardeen's 1956 Nobel Prize lecture, but even that left out important details.

Of course, this wouldn't be a *Spectrum* special issue if we didn't tell you how the original point-contact transistor worked, something that even the inventors seemed a little fuzzy on. According to our editorial director for content development, Glenn Zorpette, the best explanation of the point-contact transistor is in Bardeen's 1956 Nobel Prize lecture, but even that left out important details, which Zorpette explores in classic *Spectrum* style in "*The First Transistor* and How It Worked," on page 24.

And while we're celebrating this historic accomplishment, Senior Editor Samuel K. Moore, who covers semiconductors for *Spectrum* and curated this special issue, looks at what the transistor might be like when it turns 100. For "The Transistor of 2047" [p. 38], Moore talked to the leading lights of semiconductor engineering, many of them IEEE Fellows, to get a glimpse of a future where transistors are stacked on top of each other and are made of increasingly exotic 2D materials, even as the OG of transistor materials, germanium, is poised for a comeback.

When I was talking to Moore a few weeks ago about this issue, he mentioned that he's attending his favorite conference just as this issue comes out, the 68th edition of IEEE's International Electron Devices Meeting, in San Francisco. The mind-bending advances that emerge from that conference always get him excited about the engineering feats occurring in today's labs and on tomorrow's production lines. This year he's most excited about new devices that combine computing capability with memory to speed machine learning. Who knows, maybe the transistor of 2047 will make its debut there, too. ■

PORTRAIT BY SERGIO ALBIAC; RANDI KLETT

IEEE Spectrum, December 2022

Moore’s Law

- Moore's law is the observation that the number of transistors in a dense integrated circuit doubles about every two years. The observation is named after Gordon Moore, the co-founder of Fairchild Semiconductor and Intel, whose 1965 paper described a doubling every year in the number of components per integrated circuit, and projected this rate of growth would continue for at least another decade. In 1975, looking forward to the next decade, he revised the forecast to doubling every two years. The period is often quoted as 18 months because of a prediction by Intel executive David House (being a combination of the effect of more transistors and the transistors being faster).

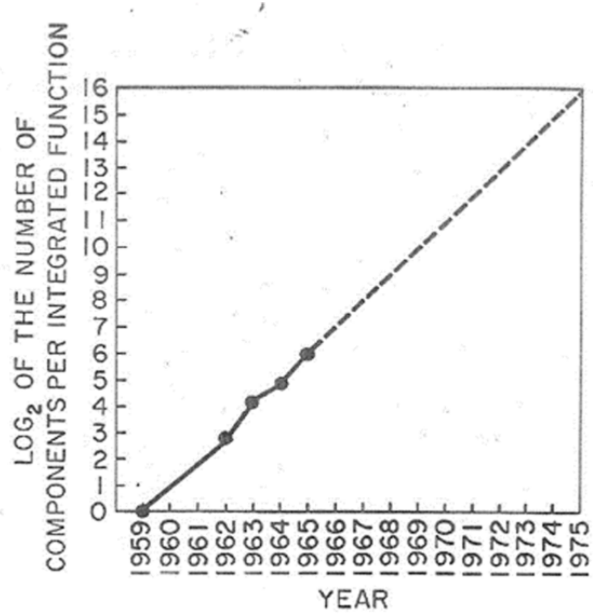
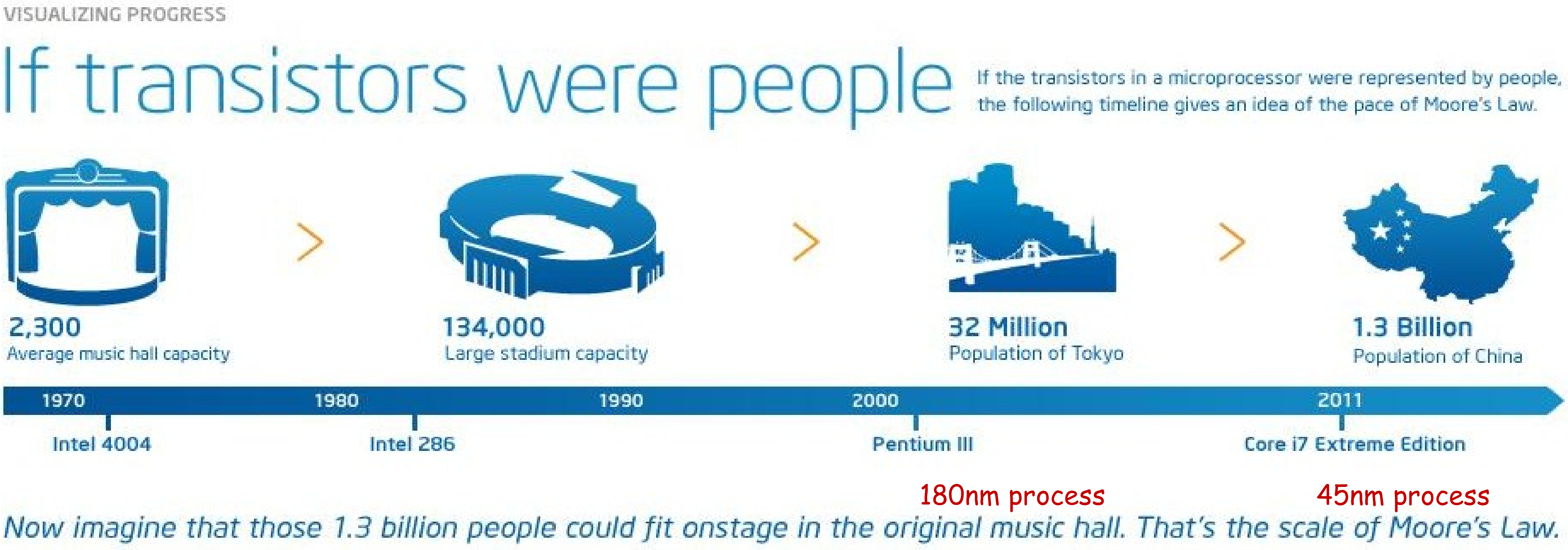


Fig. 2 Number of components per integrated function for minimum cost per component extrapolated vs time.

WIKIPEDIA

Moore's Law



VISUALIZING PROGRESS

If transistors were people

If the transistors in a microprocessor were represented by people, the following timeline gives an idea of the pace of Moore's Law.

2,300 Average music hall capacity

134,000 Large stadium capacity

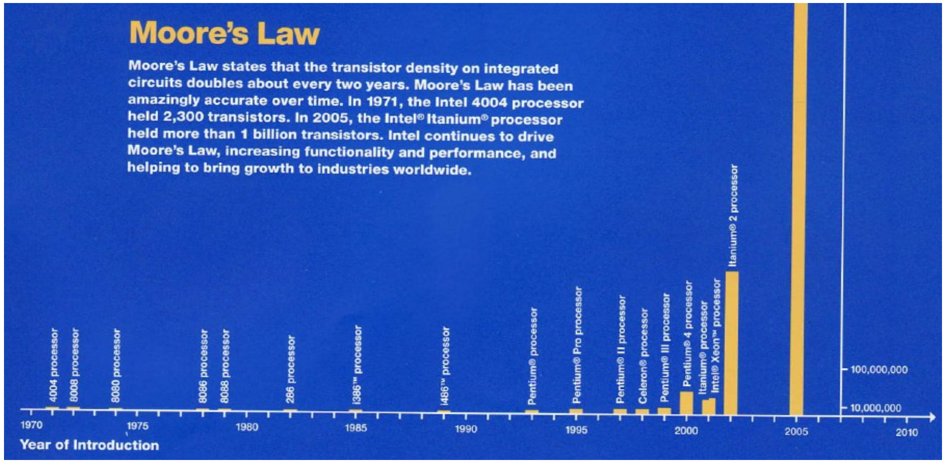
32 Million Population of Tokyo

1.3 Billion Population of China

180nm process

45nm process

Now imagine that those 1.3 billion people could fit onstage in the original music hall. That's the scale of Moore's Law.



Moore's Law

Moore's Law states that the transistor density on integrated circuits doubles about every two years. Moore's Law has been amazingly accurate over time. In 1971, the Intel 4004 processor held 2,300 transistors. In 2005, the Intel® Itanium® processor held more than 1 billion transistors. Intel continues to drive Moore's Law, increasing functionality and performance, and helping to bring growth to industries worldwide.

Moore's Law

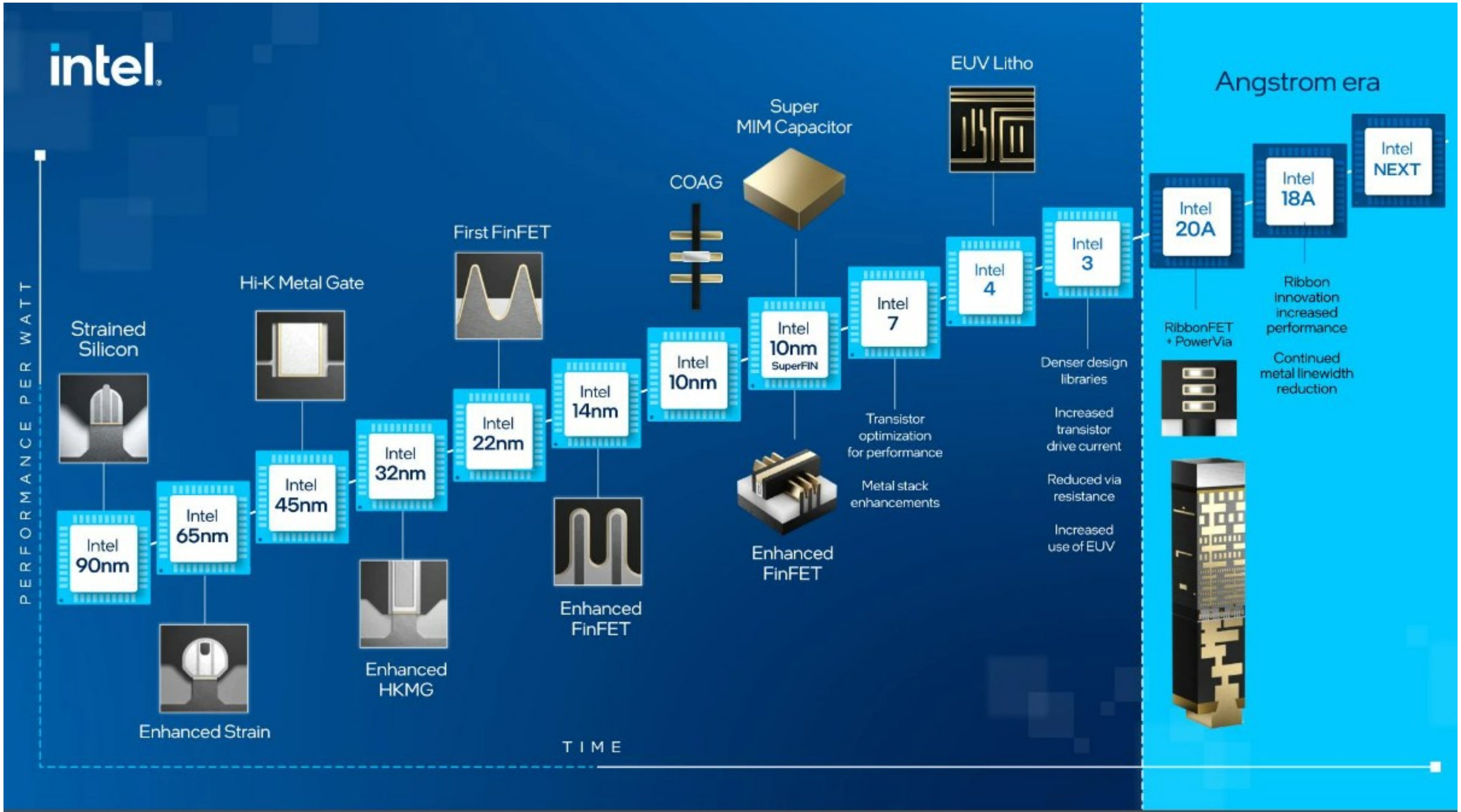


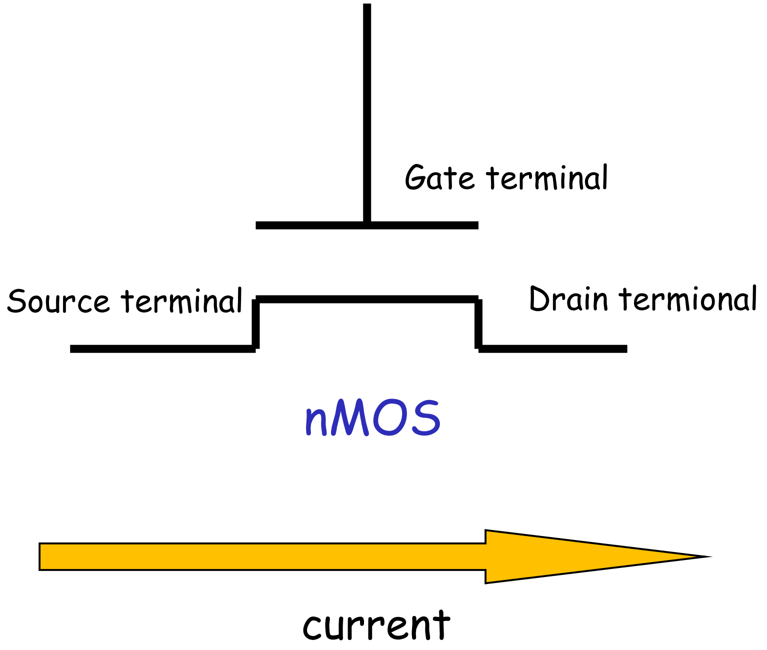
Figure 2: Transistor innovations over time

https://www.intel.com/content/www/us/en/newsroom/opinion/moore-law-now-and-in-the-future.html

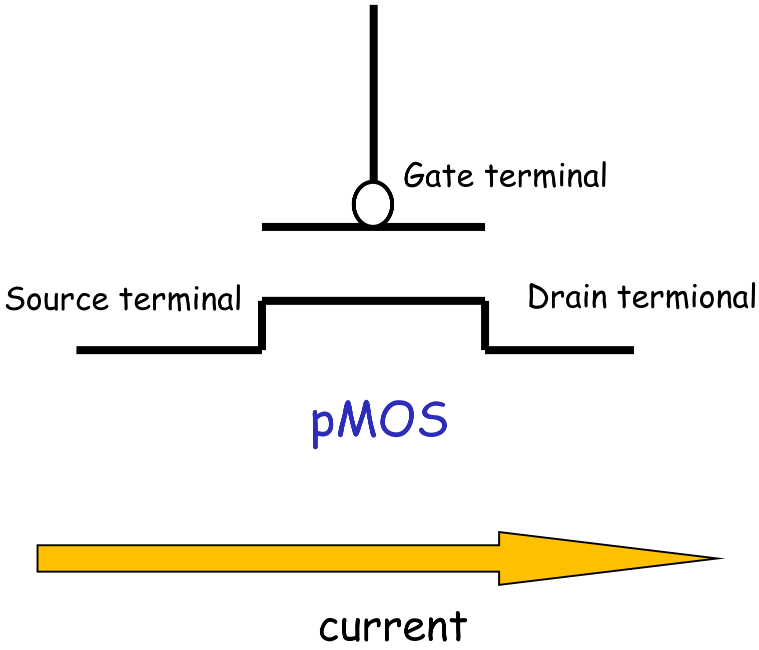
Transistor

- In an nMOS transistor, when a positive voltage is applied to the gate terminal relative to the source terminal, it creates an electric field that attracts electrons towards the gate. This forms a conductive channel between the source and drain terminals allowing current to flow through.

turn on
when a positive voltage is applied



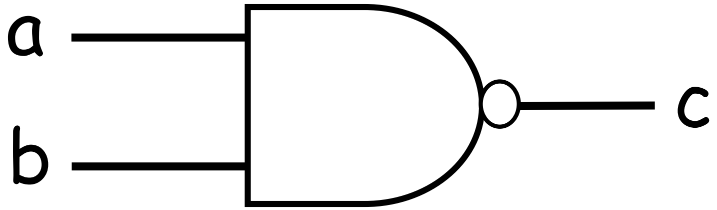
turn on
when a negative voltage is applied



Transistor and Gate

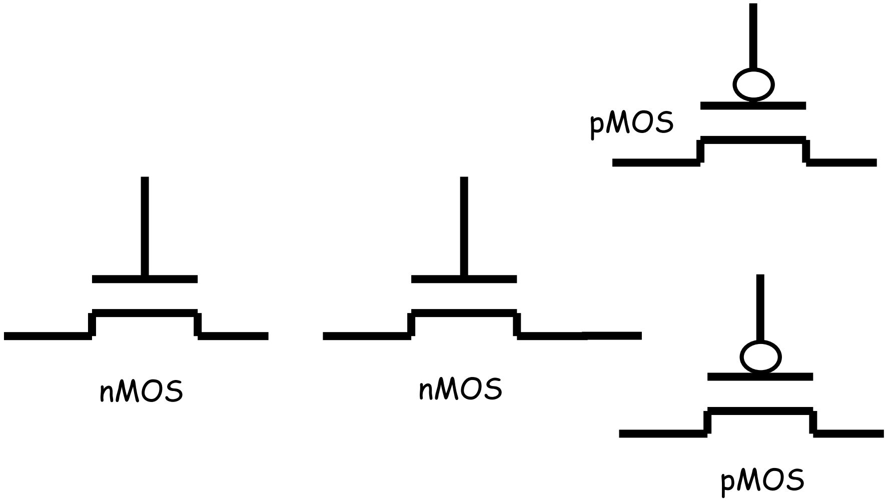
- A logic gate is a device that performs a logical operation on one or two binary inputs that produces a single binary output

NAND gate



Truth table of NAND gate

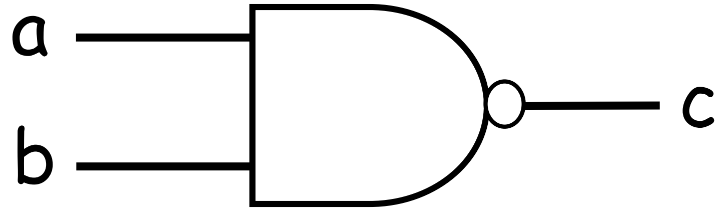
a	b	c
0	0	1
1	0	1
0	1	1
1	1	0



Transistor and Gate

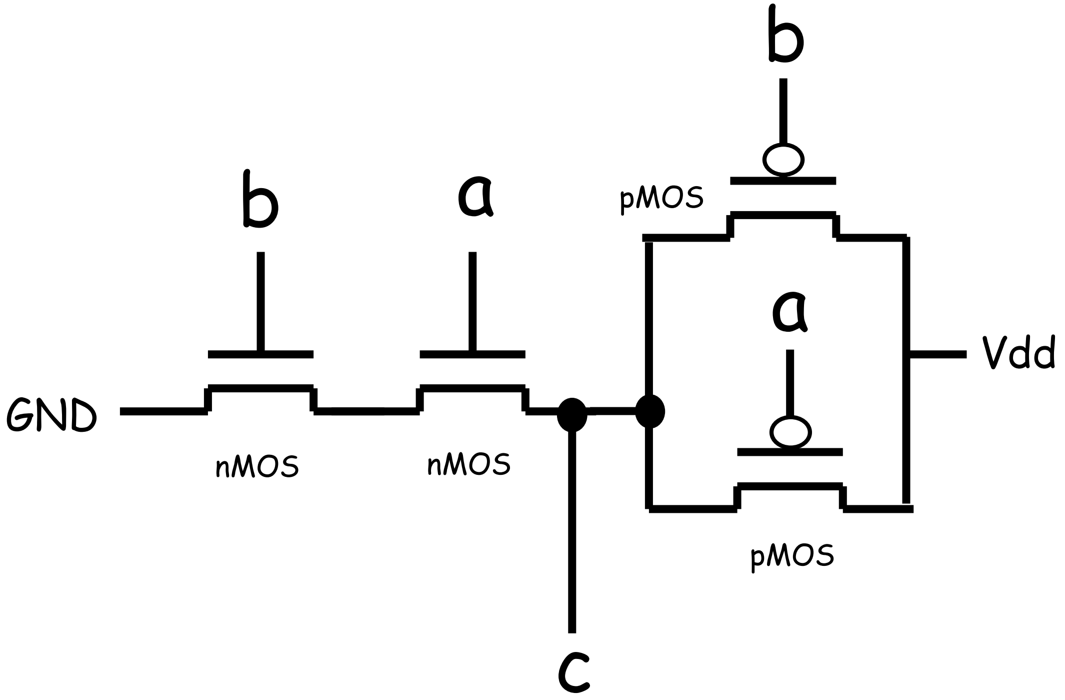
- A logic gate is a device that performs a logical operation on one or two binary inputs that produces a single binary output

NAND gate



Truth table of NAND gate

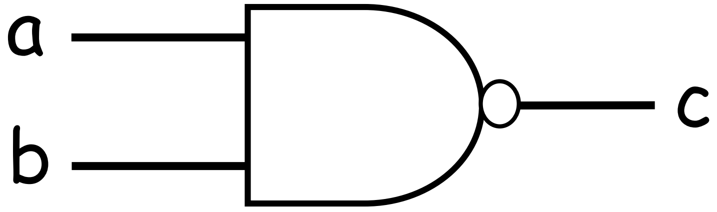
a	b	c
0	0	1
1	0	1
0	1	1
1	1	0



Transistor and Gate

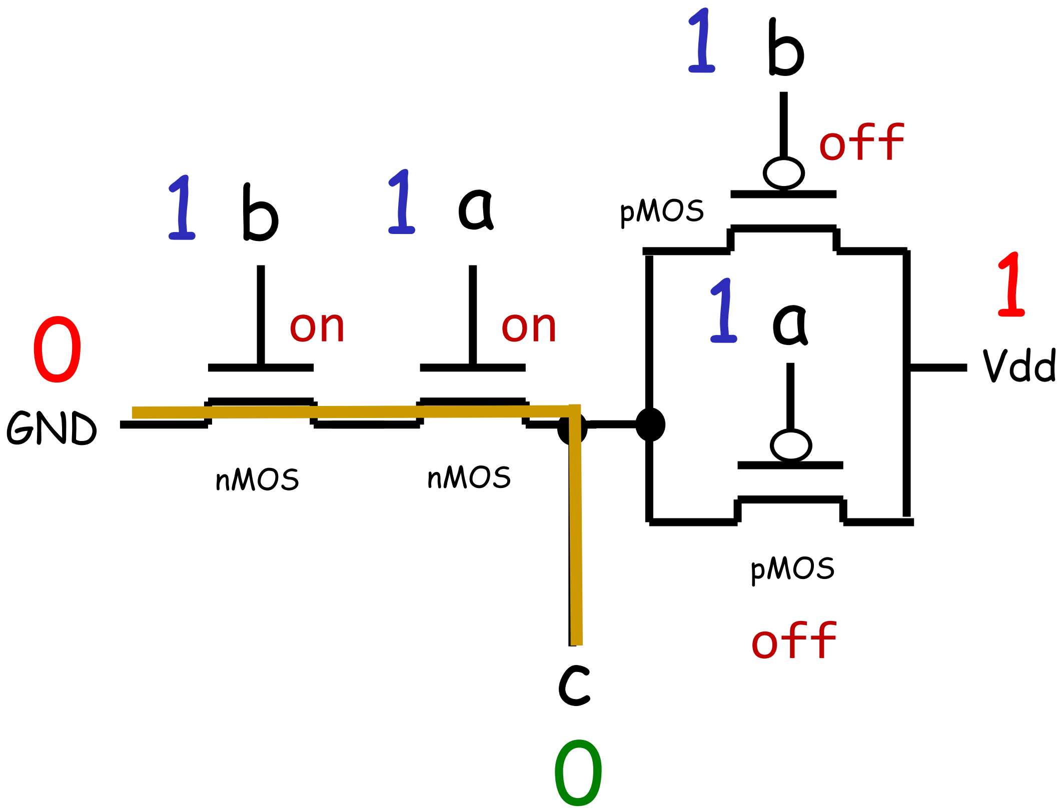
- A logic gate is a device that performs a logical operation on one or two binary inputs that produces a single binary output

NAND gate



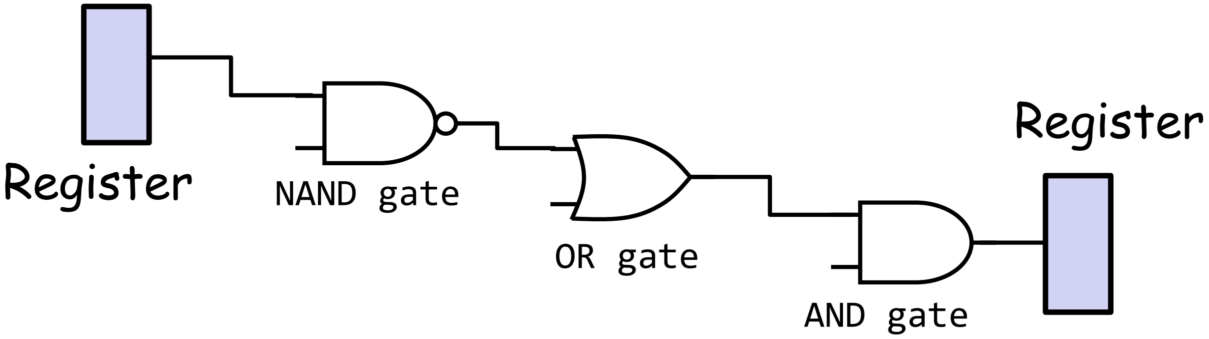
Truth table of NAND gate

a	b	c
0	0	1
1	0	1
0	1	1
1	1	0



Clock rate F is mainly determined by

- Switching speed of gates (transistors)
- The number of levels of gates
  - The maximum number of gates cascaded in series in any combinational logics.
  - In this example, the number of levels of gates is 3.
- Wiring delay and fanout
  - fan-out is the number of gate inputs driven by the output of another single logic gate.



Pollack's Rule

- Pollack's Rule states that microprocessor performance increase due to microarchitecture advances is roughly proportional to the square root of the increase in complexity. Complexity in this context means processor logic, i.e. the number of transistors.

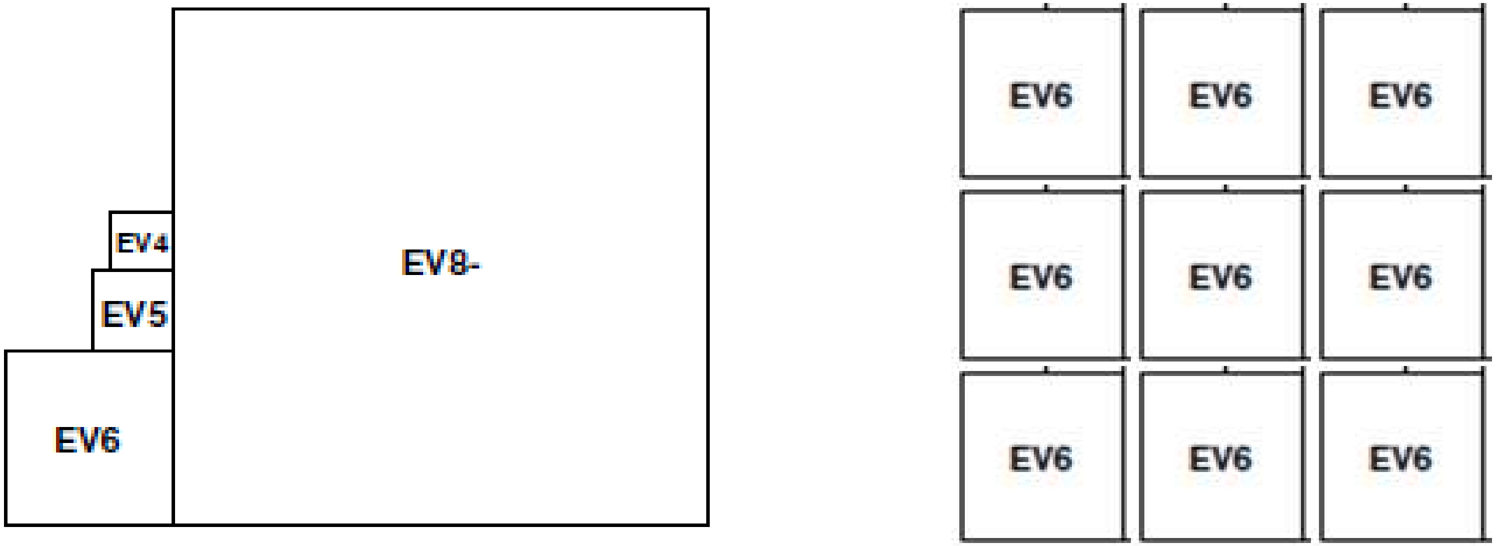
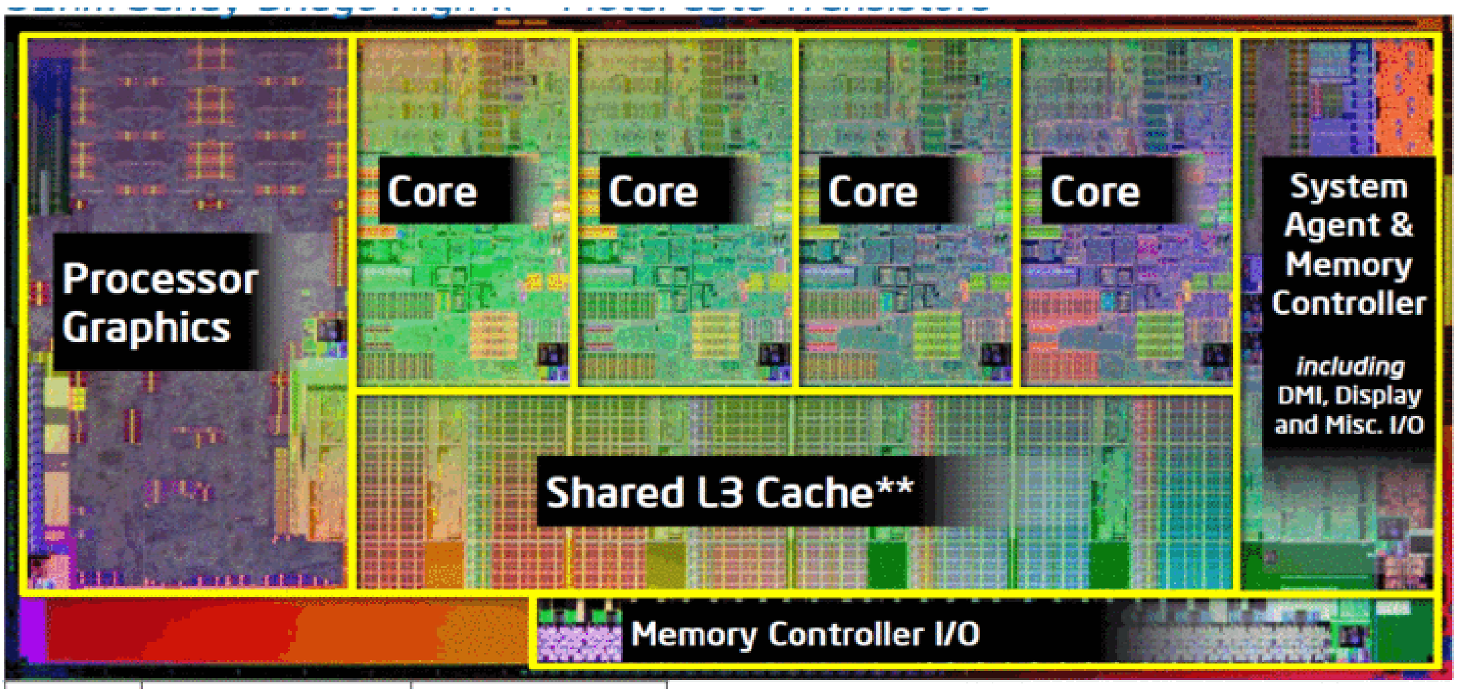


Figure 1. Relative sizes of the cores used in the study

Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction, MICRO-36

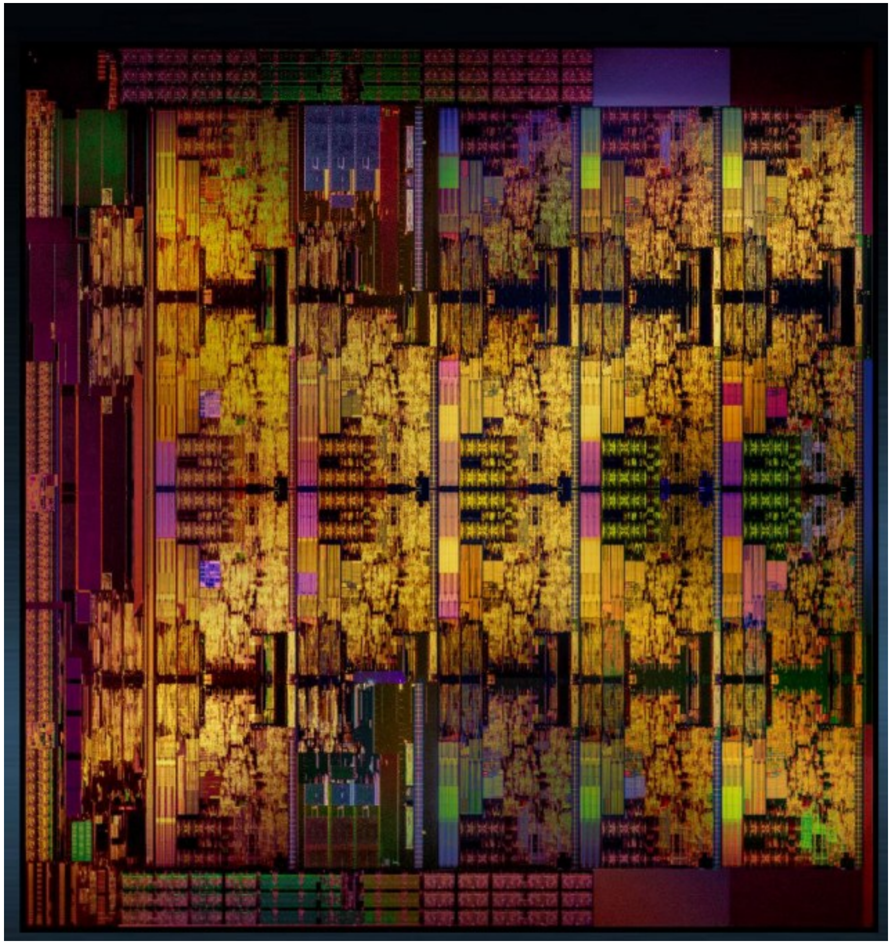
Intel Sandy Bridge, January 2011

- 4 core



Intel Skylake-X, Core i9-7980XE, 2017

- 18 core



From single-core, multi-core era to many-core era

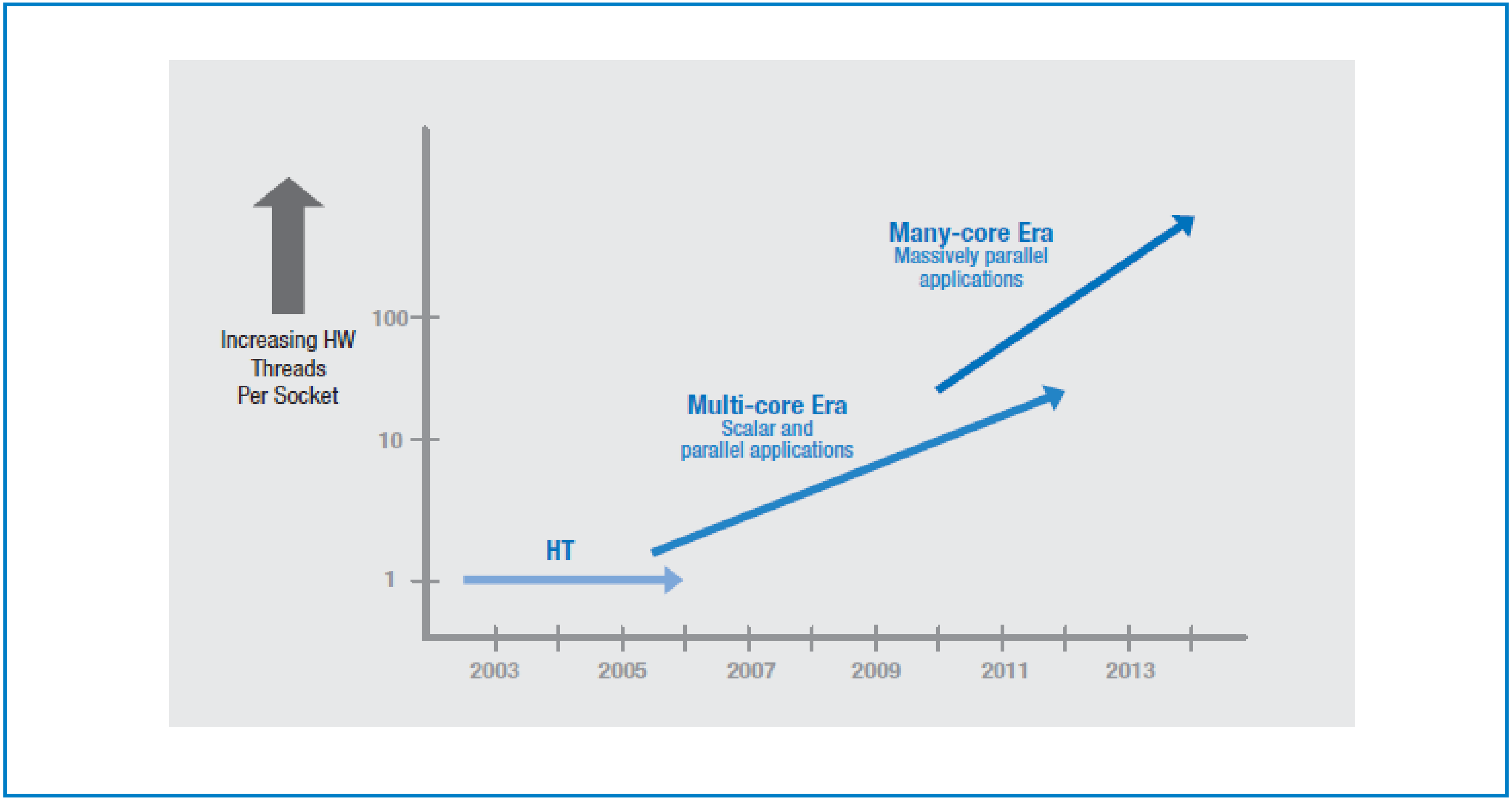


Figure 1: Current and expected eras of Intel® processor architectures

Platform 2015: Intel® Processor and Platform Evolution for the Next Decade, 2005

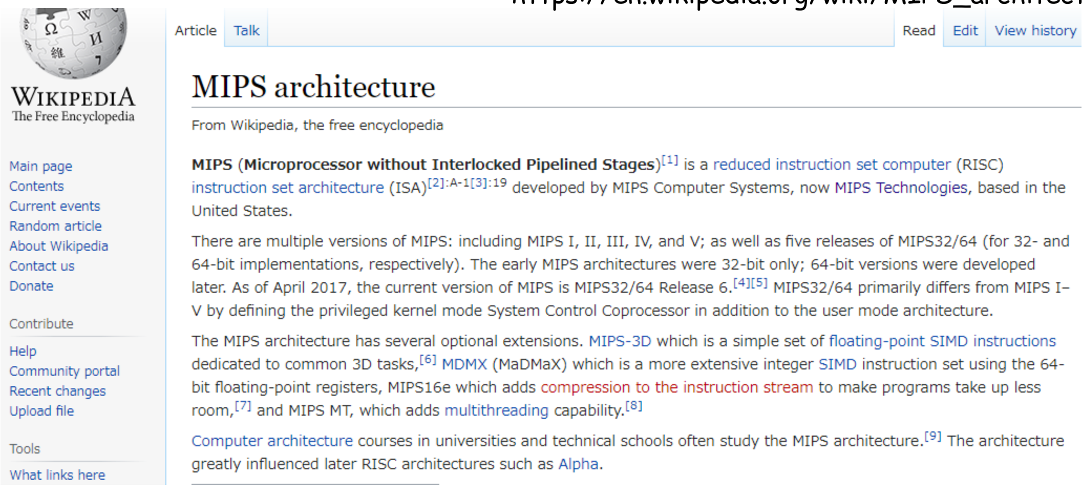
Two major ISA types: RISC vs CISC

An Instruction Set Architecture (ISA) is part of the abstract model of a computer that defines how the processor is controlled by the software. The ISA acts as an interface between the hardware and the software.

- RISC (Reduced Instruction Set Computer) philosophy
  - fixed instruction lengths
  - load-store instruction sets
  - limited addressing modes
  - limited operations
  - RISC: MIPS, Alpha, ARM, RISC-V, …
- CISC (Complex Instruction Set Computer) philosophy
  - ! fixed instruction lengths
  - ! load-store instruction sets
  - ! limited addressing modes
  - ! limited operations
  - CISC : DEC VAX11, Intel 80x86, …

MIPS, ARM, and RISC-V

https://en.wikipedia.org/wiki/MIPS_architecture



MIPS architecture

From Wikipedia, the free encyclopedia

MIPS (**Microprocessor without Interlocked Pipelined Stages**)[1] is a reduced instruction set computer (RISC) instruction set architecture (ISA)[2]:A-1[3]:19 developed by MIPS Computer Systems, now MIPS Technologies, based in the United States.

There are multiple versions of MIPS: including MIPS I, II, III, IV, and V; as well as five releases of MIPS32/64 (for 32- and 64-bit implementations, respectively). The early MIPS architectures were 32-bit only; 64-bit versions were developed later. As of April 2017, the current version of MIPS is MIPS32/64 Release 6.[4][5] MIPS32/64 primarily differs from MIPS I–V by defining the privileged kernel mode System Control Coprocessor in addition to the user mode architecture.

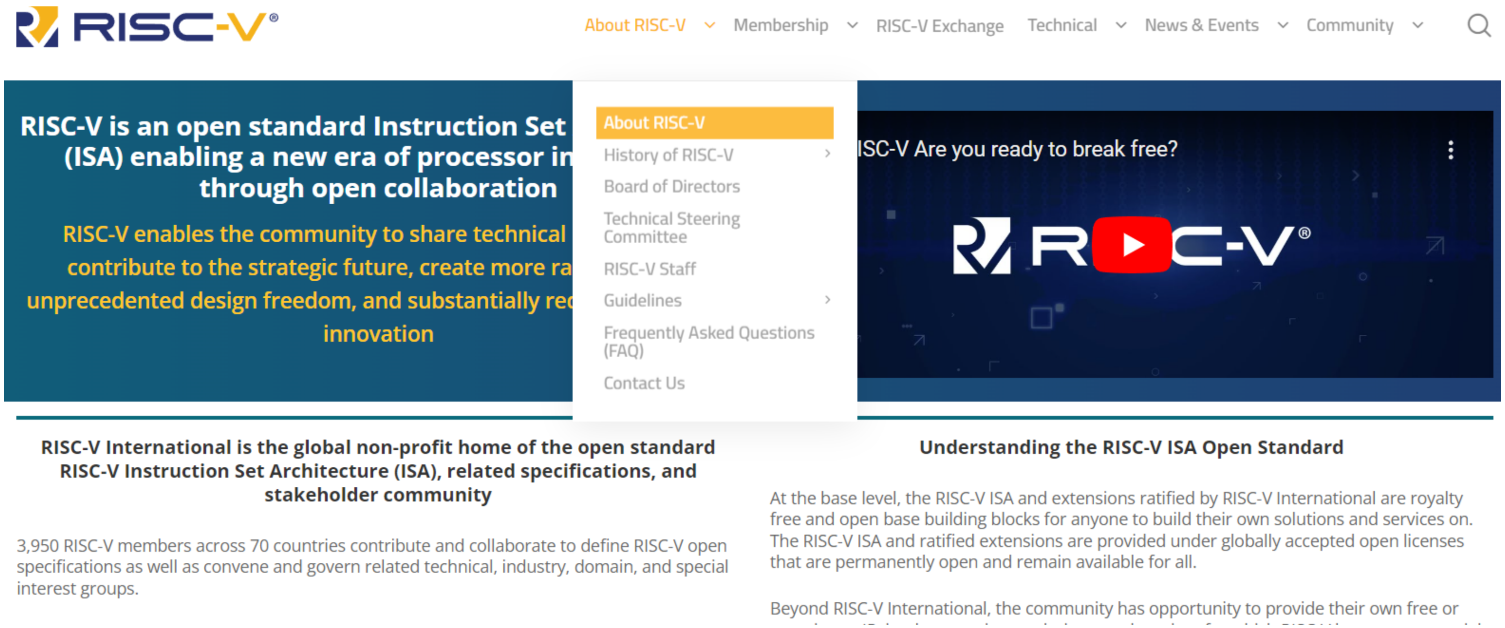
The MIPS architecture has several optional extensions. MIPS-3D which is a simple set of floating-point SIMD instructions dedicated to common 3D tasks,[6] MDMX (MaDMaX) which is a more extensive integer SIMD instruction set using the 64-bit floating-point registers, MIPS16e which adds compression to the instruction stream to make programs take up less room,[7] and MIPS MT, which adds multithreading capability.[8]

Computer architecture courses in universities and technical schools often study the MIPS architecture.[9] The architecture greatly influenced later RISC architectures such as Alpha.

arm

Arm "ABCD" building in Cherry Hinton, Cambridge, UK

ARM (Advanced RISC Machine)



RISC-V is an open standard Instruction Set (ISA) enabling a new era of processor in through open collaboration

RISC-V enables the community to share technical contribute to the strategic future, create more ra unprecedented design freedom, and substantially re innovation

RISC-V International is the global non-profit home of the open standard RISC-V Instruction Set Architecture (ISA), related specifications, and stakeholder community

3,950 RISC-V members across 70 countries contribute and collaborate to define RISC-V open specifications as well as convene and govern related technical, industry, domain, and special interest groups.

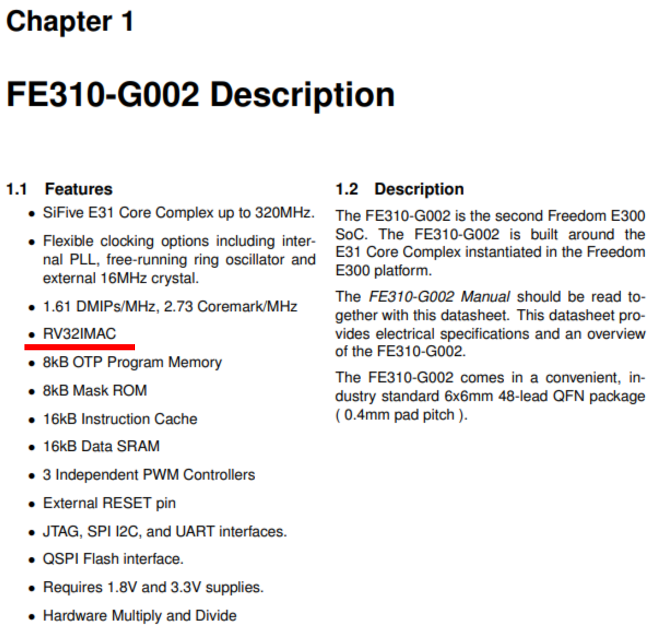
Understanding the RISC-V ISA Open Standard

At the base level, the RISC-V ISA and extensions ratified by RISC-V International are royalty free and open base building blocks for anyone to build their own solutions and services on. The RISC-V ISA and ratified extensions are provided under globally accepted open licenses that are permanently open and remain available for all.

Beyond RISC-V International, the community has opportunity to provide their own free or

https://riscv.org/

RISC-V base and extensions



Chapter 1

FE310-G002 Description

1.1 Features

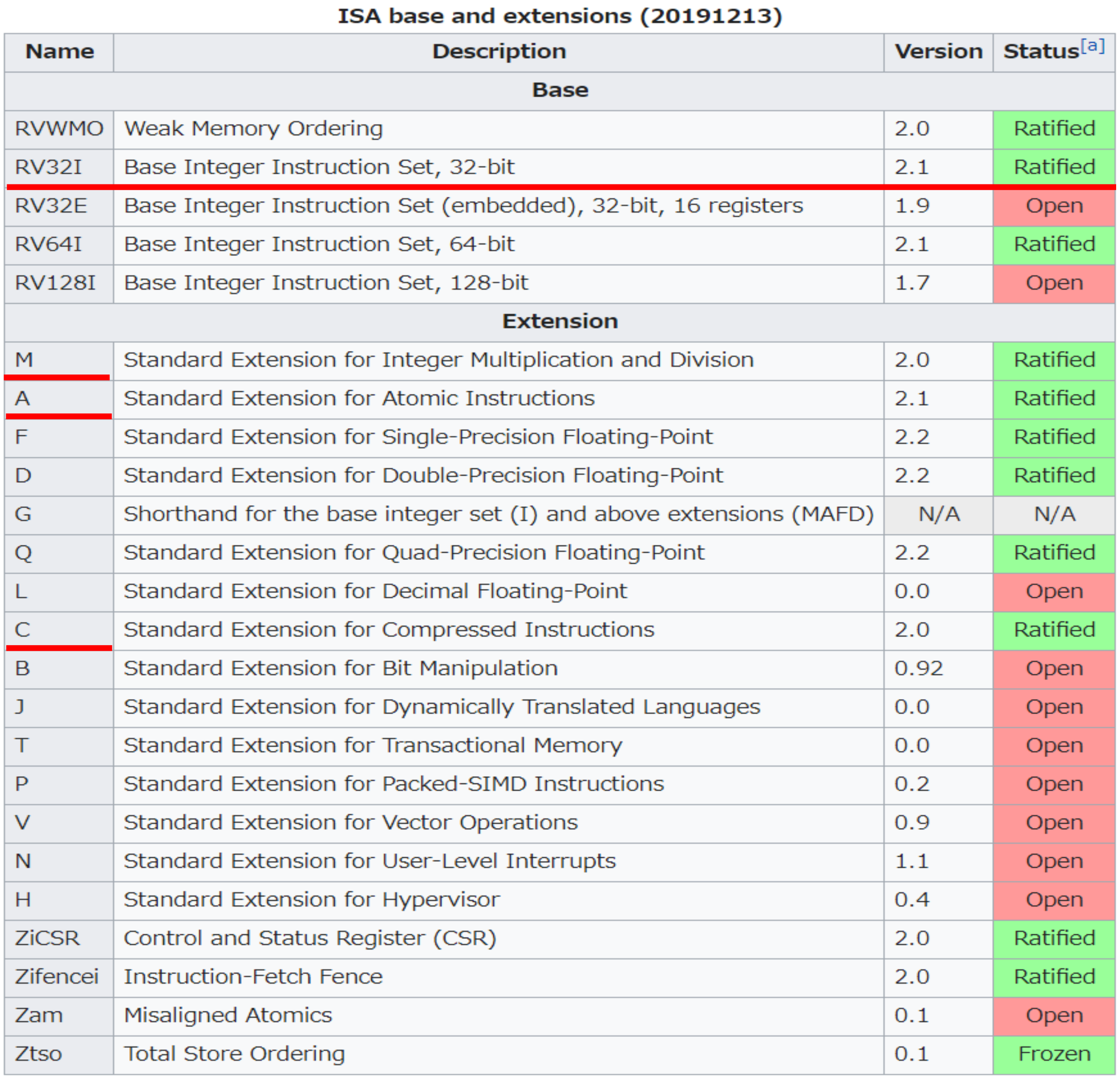
- SiFive E31 Core Complex up to 320MHz.
- Flexible clocking options including internal PLL, free-running ring oscillator and external 16MHz crystal.
- 1.61 DMIPs/MHz, 2.73 Coremark/MHz
- RV32IMAC
- 8kB OTP Program Memory
- 8kB Mask ROM
- 16kB Instruction Cache
- 16kB Data SRAM
- 3 Independent PWM Controllers
- External RESET pin
- JTAG, SPI I2C, and UART interfaces.
- QSPI Flash interface.
- Requires 1.8V and 3.3V supplies.
- Hardware Multiply and Divide

1.2 Description

The FE310-G002 is the second Freedom E300 SoC. The FE310-G002 is built around the E31 Core Complex instantiated in the Freedom E300 platform.

The *FE310-G002 Manual* should be read together with this datasheet. This datasheet provides electrical specifications and an overview of the FE310-G002.

The FE310-G002 comes in a convenient, industry standard 6x6mm 48-lead QFN package (0.4mm pad pitch).

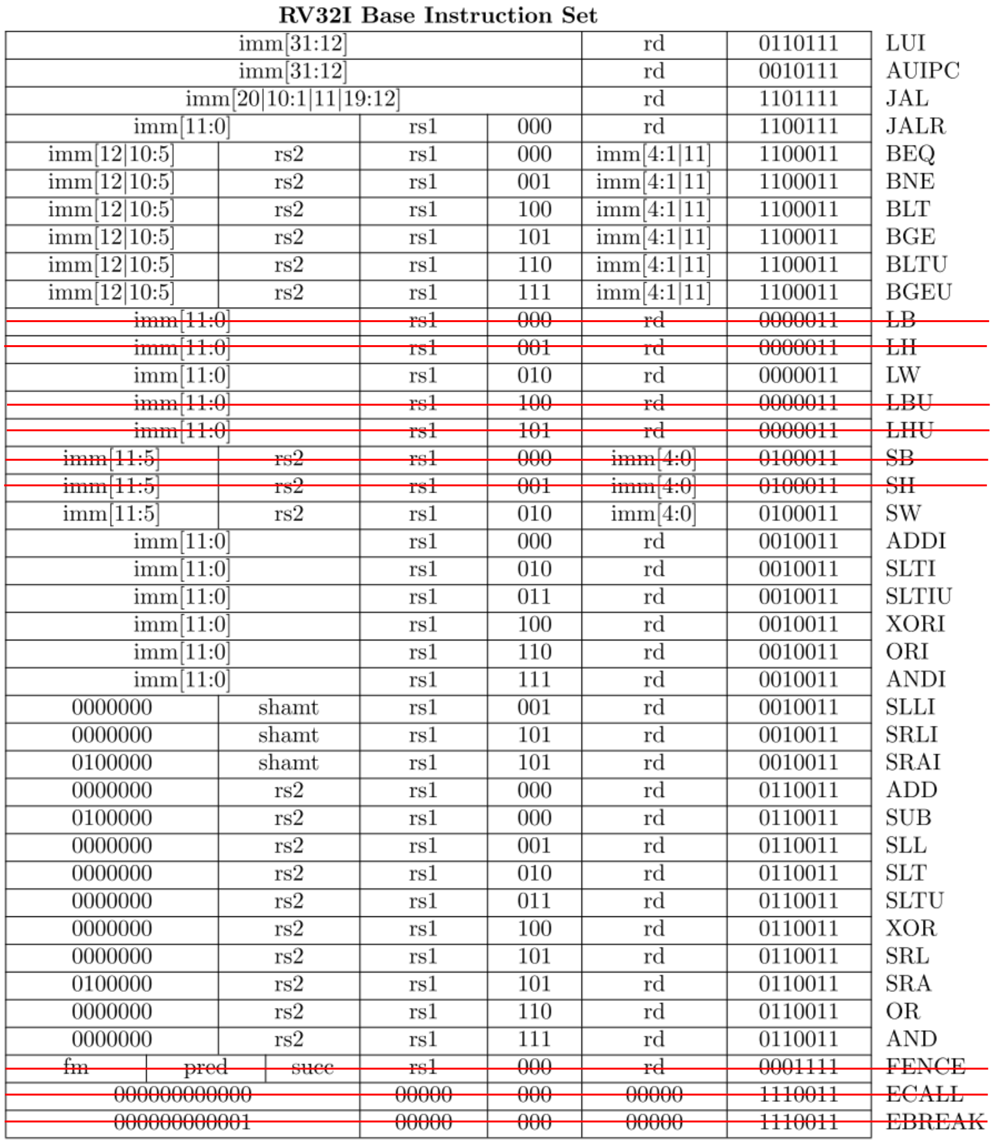


ISA base and extensions (20191213)

Name	Description	Version	Status[a]
Base			
RVWMO	Weak Memory Ordering	2.0	Ratified
RV32I	Base Integer Instruction Set, 32-bit	2.1	Ratified
RV32E	Base Integer Instruction Set (embedded), 32-bit, 16 registers	1.9	Open
RV64I	Base Integer Instruction Set, 64-bit	2.1	Ratified
RV128I	Base Integer Instruction Set, 128-bit	1.7	Open
Extension			
M	Standard Extension for Integer Multiplication and Division	2.0	Ratified
A	Standard Extension for Atomic Instructions	2.1	Ratified
F	Standard Extension for Single-Precision Floating-Point	2.2	Ratified
D	Standard Extension for Double-Precision Floating-Point	2.2	Ratified
G	Shorthand for the base integer set (I) and above extensions (MAFD)	N/A	N/A
Q	Standard Extension for Quad-Precision Floating-Point	2.2	Ratified
L	Standard Extension for Decimal Floating-Point	0.0	Open
C	Standard Extension for Compressed Instructions	2.0	Ratified
B	Standard Extension for Bit Manipulation	0.92	Open
J	Standard Extension for Dynamically Translated Languages	0.0	Open
T	Standard Extension for Transactional Memory	0.0	Open
P	Standard Extension for Packed-SIMD Instructions	0.2	Open
V	Standard Extension for Vector Operations	0.9	Open
N	Standard Extension for User-Level Interrupts	1.1	Open
H	Standard Extension for Hypervisor	0.4	Open
ZiCSR	Control and Status Register (CSR)	2.0	Ratified
Zifencei	Instruction-Fetch Fence	2.0	Ratified
Zam	Misaligned Atomics	0.1	Open
Ztso	Total Store Ordering	0.1	Frozen

RISC-V RV32I base and our target instructions

We do not support some system instructions (FENCE, ECALL, EBREAK) and 8-bit or 16-bit loads (LB, LH, LBU, LHU) and stores (SB, SH) of RV32I.

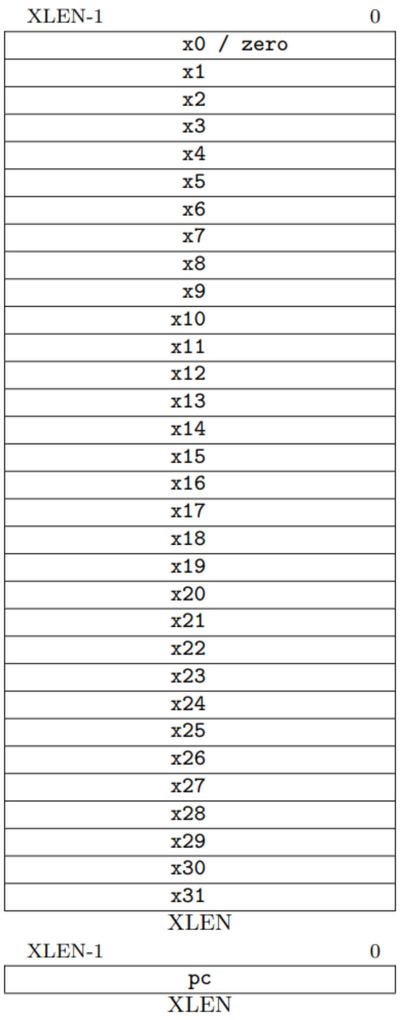


RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
~~imm[11:0]~~		~~rs1~~	~~000~~	~~rd~~	~~0000011~~	~~LB~~
~~imm[11:0]~~		~~rs1~~	~~001~~	~~rd~~	~~0000011~~	~~LH~~
imm[11:0]		rs1	010	rd	0000011	LW
~~imm[11:0]~~		~~rs1~~	~~100~~	~~rd~~	~~0000011~~	~~LBU~~
~~imm[11:0]~~		~~rs1~~	~~101~~	~~rd~~	~~0000011~~	~~LHU~~
~~imm[11:5]~~	~~rs2~~	~~rs1~~	~~000~~	~~imm[4:0]~~	~~0100011~~	~~SB~~
~~imm[11:5]~~	~~rs2~~	~~rs1~~	~~001~~	~~imm[4:0]~~	~~0100011~~	~~SH~~
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
~~fm~~	~~pred~~ ~~succ~~	~~rs1~~	~~000~~	~~rd~~	~~0001111~~	~~FENCE~~
~~000000000000~~		~~00000~~	~~000~~	~~00000~~	~~1110011~~	~~ECALL~~
~~000000000001~~		~~00000~~	~~000~~	~~00000~~	~~1110011~~	~~EBREAK~~

RISC-V general-purpose registers

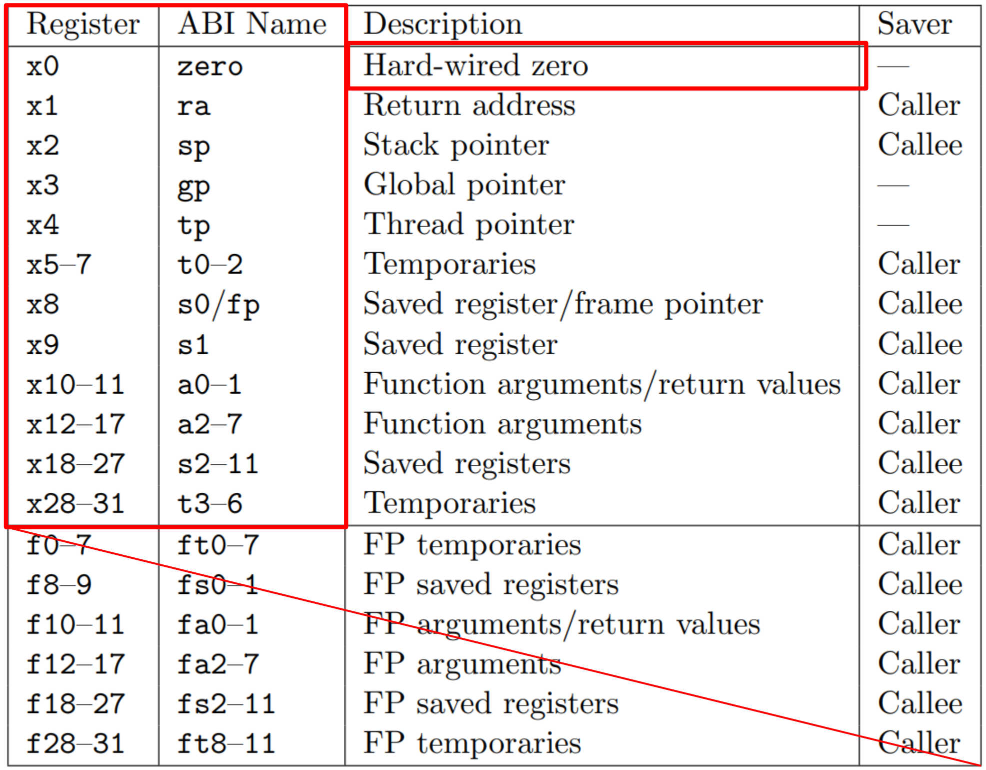
XLEN = 32
for 32bit ISA



```
XLEN-1                0
      x0 / zero
         x1
         x2
         x3
         x4
         x5
         x6
         x7
         x8
         x9
        x10
        x11
        x12
        x13
        x14
        x15
        x16
        x17
        x18
        x19
        x20
        x21
        x22
        x23
        x24
        x25
        x26
        x27
        x28
        x29
        x30
        x31
        XLEN
XLEN-1                0
         pc
        XLEN
```

Figure 2.1: RISC-V base unprivileged integer register state.

ABI(Application Binary Interface) name

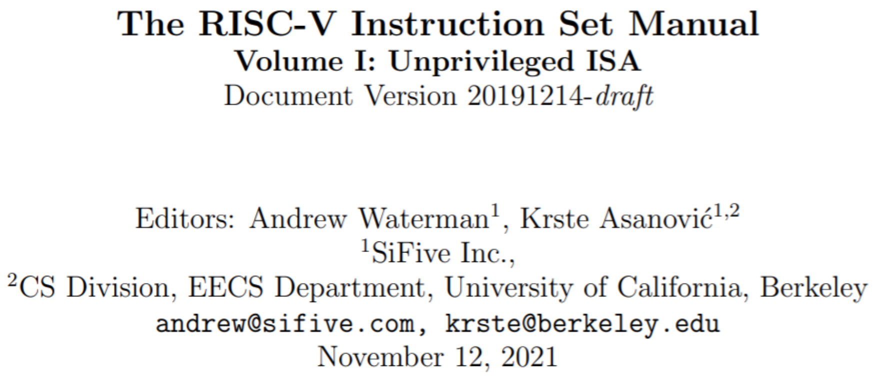


Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Table 18.2: RISC-V calling convention register usage.

RV32I does not have floating point regesters of f0 - f31.

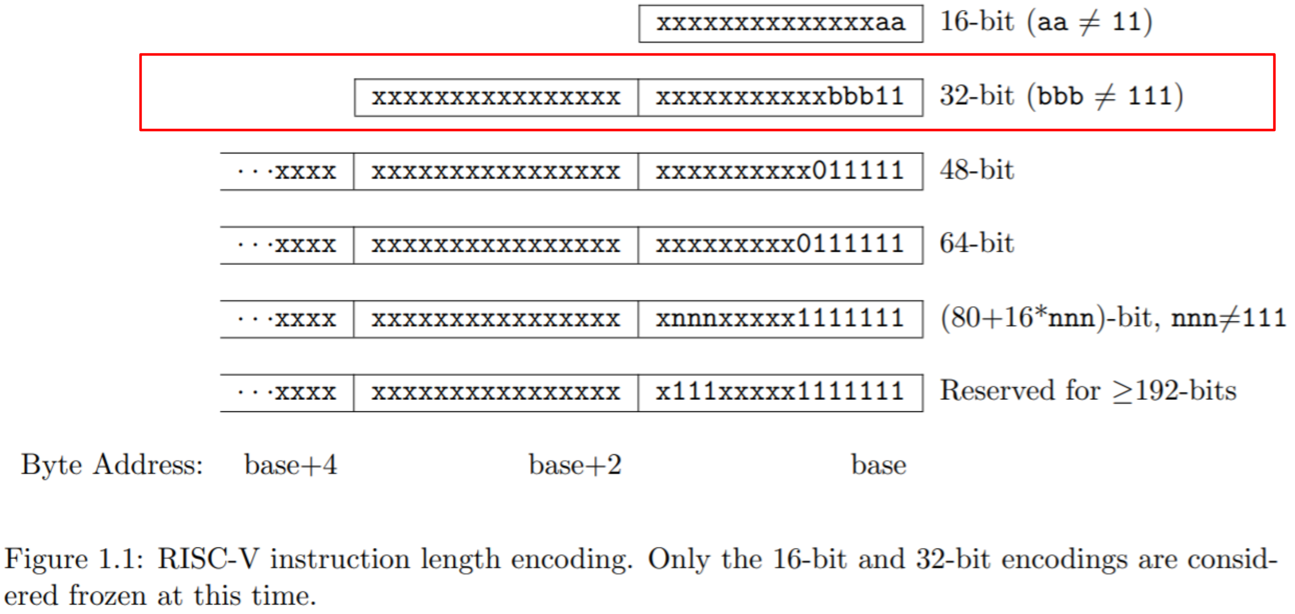
RISC-V instruction length encoding



The RISC-V Instruction Set Manual
Volume I: Unprivileged ISA
Document Version 20191214-*draft*

Editors: Andrew Waterman[1], Krste Asanović[1,2]
[1]SiFive Inc.,
[2]CS Division, EECS Department, University of California, Berkeley
`andrew@sifive.com, krste@berkeley.edu`
November 12, 2021

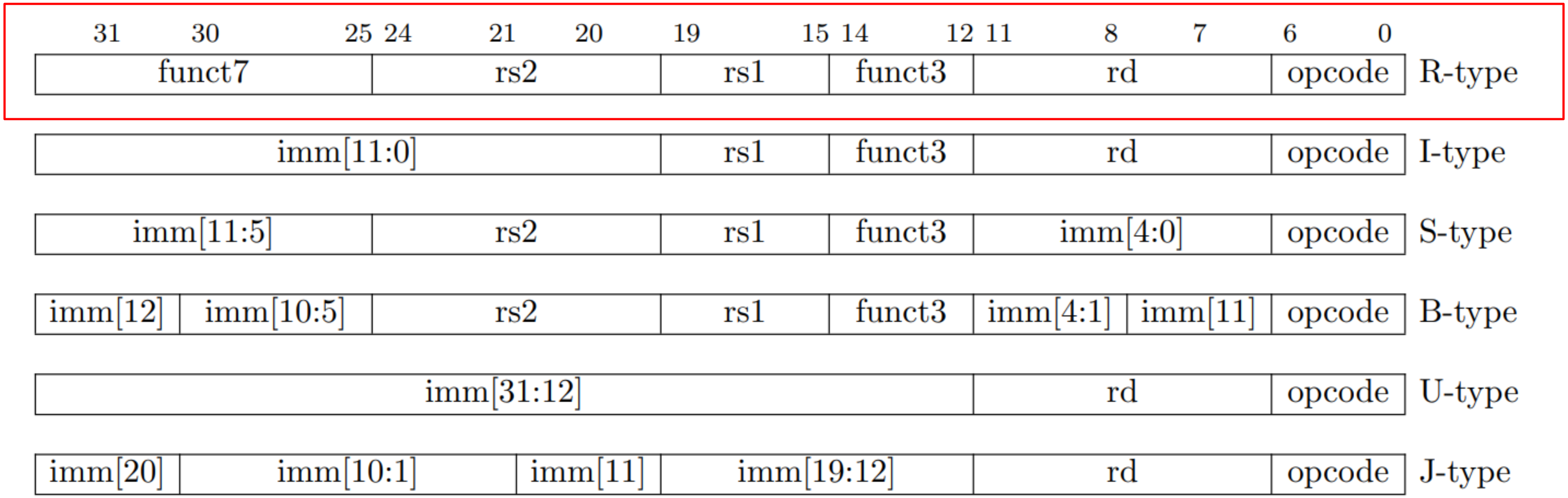
We support 32-bit length instructions. 16-bit length instructions called compressed instructions are used in some embedded systems.



			xxxxxxxxxxxxxxaa	16-bit (aa ≠ 11)
		xxxxxxxxxxxxxxxx	xxxxxxxxxxxbbb11	32-bit (bbb ≠ 111)
	···xxxx	xxxxxxxxxxxxxxxx	xxxxxxxxxx011111	48-bit
	···xxxx	xxxxxxxxxxxxxxxx	xxxxxxxxx0111111	64-bit
	···xxxx	xxxxxxxxxxxxxxxx	xnnnxxxxx1111111	(80+16*nnn)-bit, nnn≠111
	···xxxx	xxxxxxxxxxxxxxxx	x111xxxxx1111111	Reserved for ≥192-bits
Byte Address:	base+4	base+2	base	

Figure 1.1: RISC-V instruction length encoding. Only the 16-bit and 32-bit encodings are considered frozen at this time.

RISC-V base instruction format

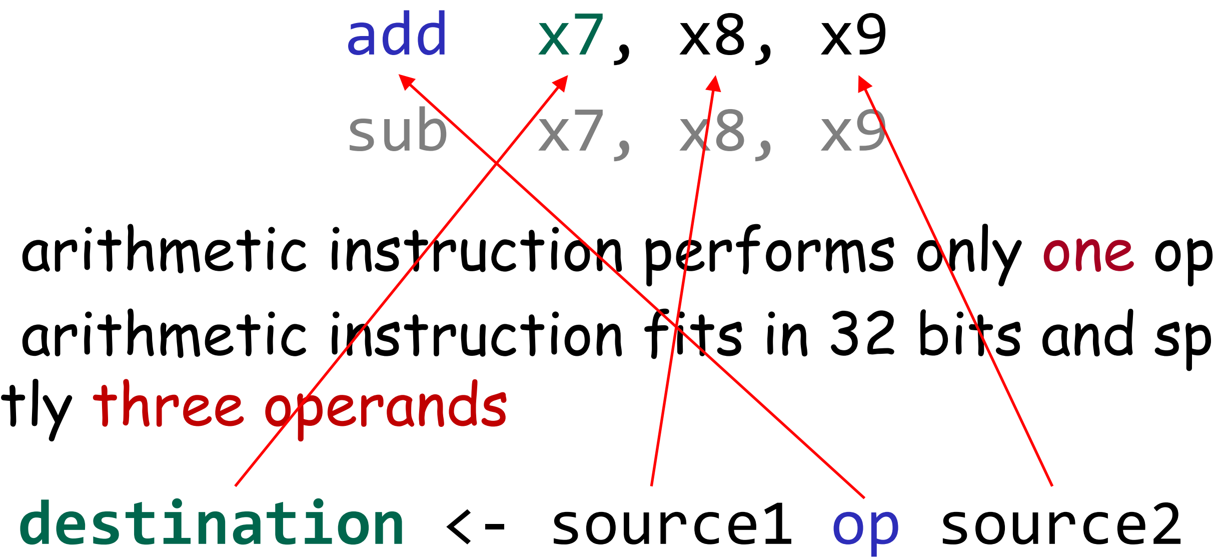


<table>
<tr><td>31</td><td>30 25</td><td>24 21</td><td>20</td><td>19 15</td><td>14 12</td><td>11 8</td><td>7</td><td>6 0</td><td></td></tr>
<tr><td colspan="2">funct7</td><td colspan="2">rs2</td><td>rs1</td><td>funct3</td><td colspan="2">rd</td><td>opcode</td><td>R-type</td></tr>
<tr><td colspan="4">imm[11:0]</td><td>rs1</td><td>funct3</td><td colspan="2">rd</td><td>opcode</td><td>I-type</td></tr>
<tr><td colspan="2">imm[11:5]</td><td colspan="2">rs2</td><td>rs1</td><td>funct3</td><td colspan="2">imm[4:0]</td><td>opcode</td><td>S-type</td></tr>
<tr><td>imm[12]</td><td>imm[10:5]</td><td colspan="2">rs2</td><td>rs1</td><td>funct3</td><td>imm[4:1]</td><td>imm[11]</td><td>opcode</td><td>B-type</td></tr>
<tr><td colspan="6">imm[31:12]</td><td colspan="2">rd</td><td>opcode</td><td>U-type</td></tr>
<tr><td>imm[20]</td><td colspan="2">imm[10:1]</td><td>imm[11]</td><td colspan="2">imm[19:12]</td><td colspan="2">rd</td><td>opcode</td><td>J-type</td></tr>
</table>

Figure 2.3: RISC-V base instruction formats showing immediate variants.

RISC-V *Arithmetic Instructions*

- RISC-V assembly language arithmetic statement



```
add   x7, x8, x9
sub   x7, x8, x9
```

- Each arithmetic instruction performs only one operation
- Each arithmetic instruction fits in 32 bits and specifies exactly three operands

```
destination <- source1 op source2
```

- Operand order is fixed (destination first)
- Those operands are all contained in the datapath's register file (x0, ..., x31)

Exercise 1

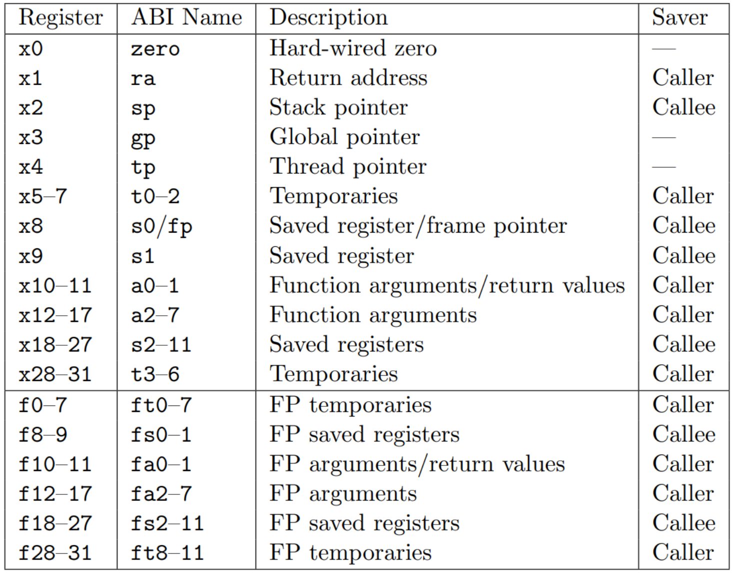
- Compiling a C assignment using registers

```
f = ( g + h ) - ( i + j );
```

- The variables f, g, h, i, and j are assigned to the registers s0, s1, s2, s3, and s4, respectively. What is the compiled RISC-V code?

```
s0 = ( s1 + s2 ) - ( s3 + s4 );
```

```
t0 = s1 + s2;
t1 = s3 + s4;
s0 = t0 - t1;
```



Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Table 18.2: RISC-V calling convention register usage.

(1) Machine Language - Add instruction (add)

- Instructions are 32 bits long
- Arithmetic Instruction Format (R-type):

`add x7, x8, x9`

funct7	rs2	rs1	funct3	rd	opcode

R-type

opcode 7-bits *opcode* that specifies the operation

rs1 5-bits register file address of the first source operand

rs2 5-bits register file address of the second source operand

rd 5-bits register file address of the result's destination

funct3 and **funct7** 10-bits select the type of operation (function)

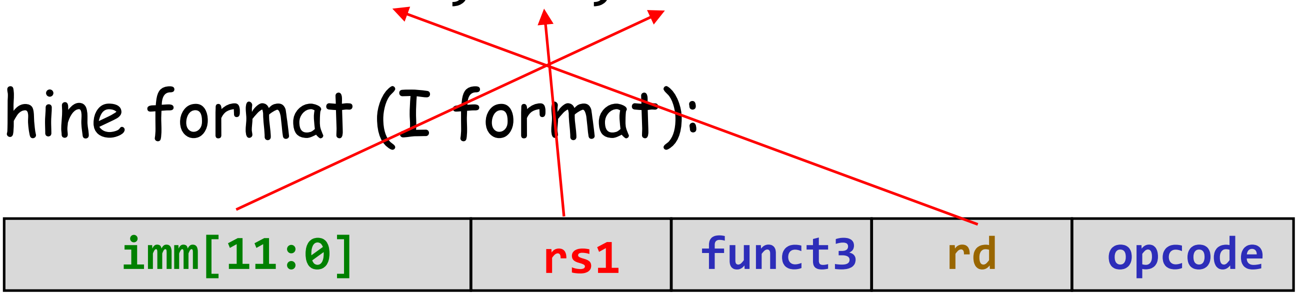
Adapted from Computer Organization and Design, Patterson & Hennessy, © 2005

(2) RISC-V Add immediate instruction (addi)

- Small constants are used often in typical code
- Possible approaches?
  - put “typical constants” in memory and load them
  - create hard-wired registers (like x0) for constants like 1
  - have special instructions that contain constants !

```
addi  x7, x8, -2     # x7 = x8 + (-2)
```

- Machine format (I format):



imm[11:0]	rs1	funct3	rd	opcode

I-type

- The constant is kept inside the instruction itself
  - Immediate format limits values to the range $+2^{11}–1$ to -2^{11}

RISC-V Memory Access Instructions

- RISC-V has two basic data transfer instructions for accessing memory
- `lw  x5, 24(x7)     # load word from memory`
- `sw  x3, 28(x9)     # store word to memory`

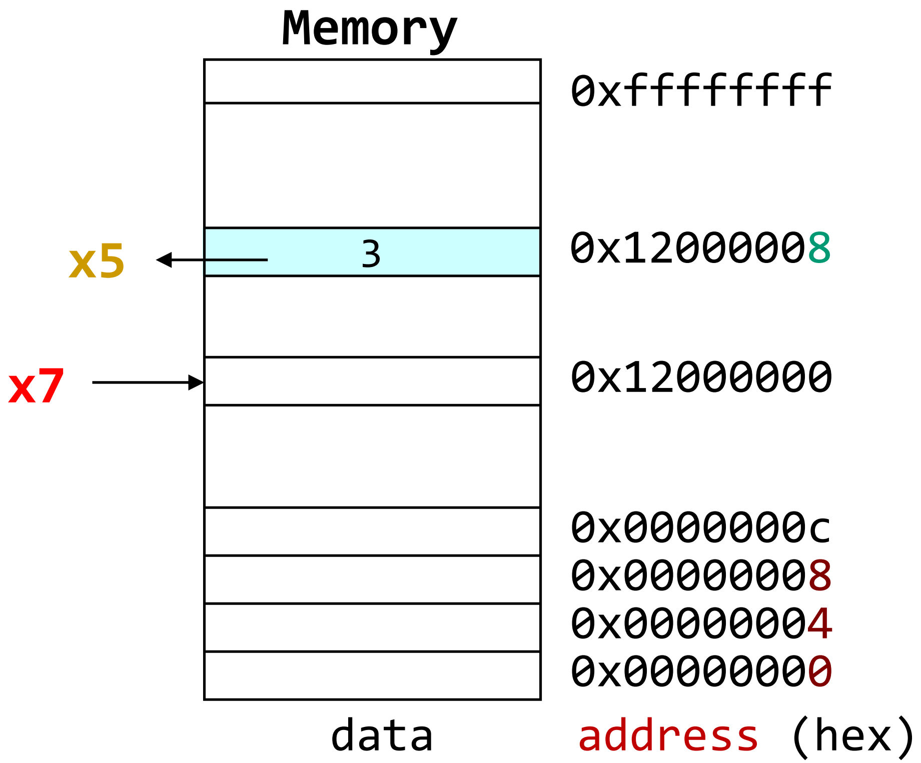
- The data is loaded into (lw) or stored from (sw) a register in the register file
- The memory address – a 32 bit address – is formed by adding the contents of the base address register to the offset value

(3) Machine Language - Load word instruction (lw)

- Load Instruction Format (I-type):

```
lw x5, 8(x7)
```

imm[11:0]	rs1	funct3	rd	opcode	I-type



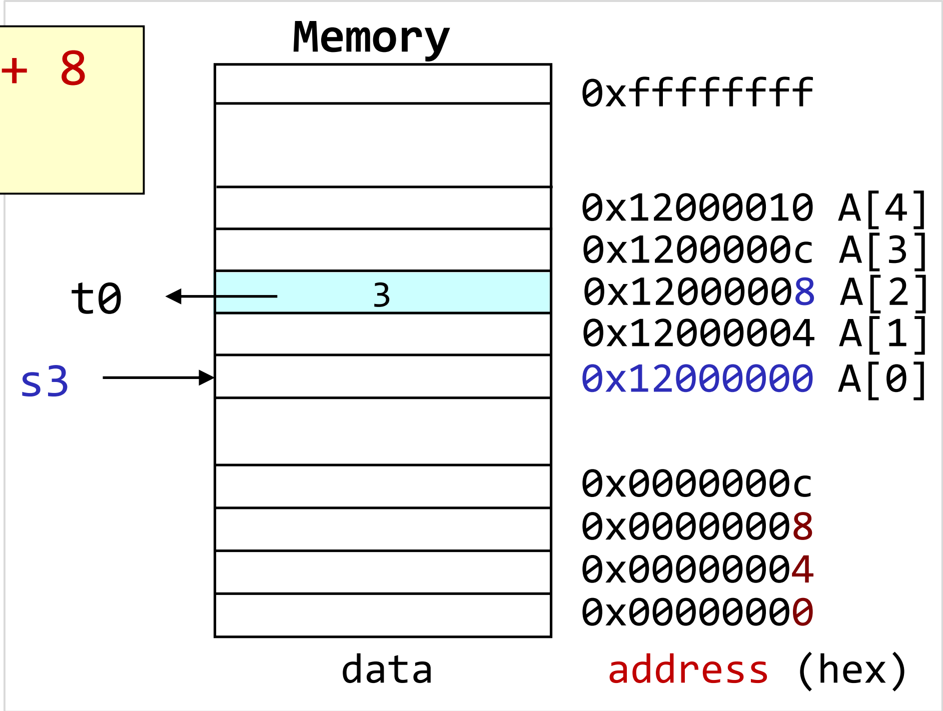
Exercise 2

- Compiling an assignment when an operand is in memory

```
g = h + A[2];
```

- Let's assume that A is an array of 100 words and the compiler has associated the variable g and h with the registers s1 and s2.
  Let's also assume that the starting address, or base address, of the array is in s3. Compile this C code.

```
t0 = A[2];  # address is s3 + 8
s1 = s2 + t0;
```



Memory

0xffffffff

0x12000010 A[4]
0x1200000c A[3]
0x12000008 A[2]
0x12000004 A[1]
0x12000000 A[0]

0x0000000c
0x00000008
0x00000004
0x00000000

t0

3

s3

data

address (hex)

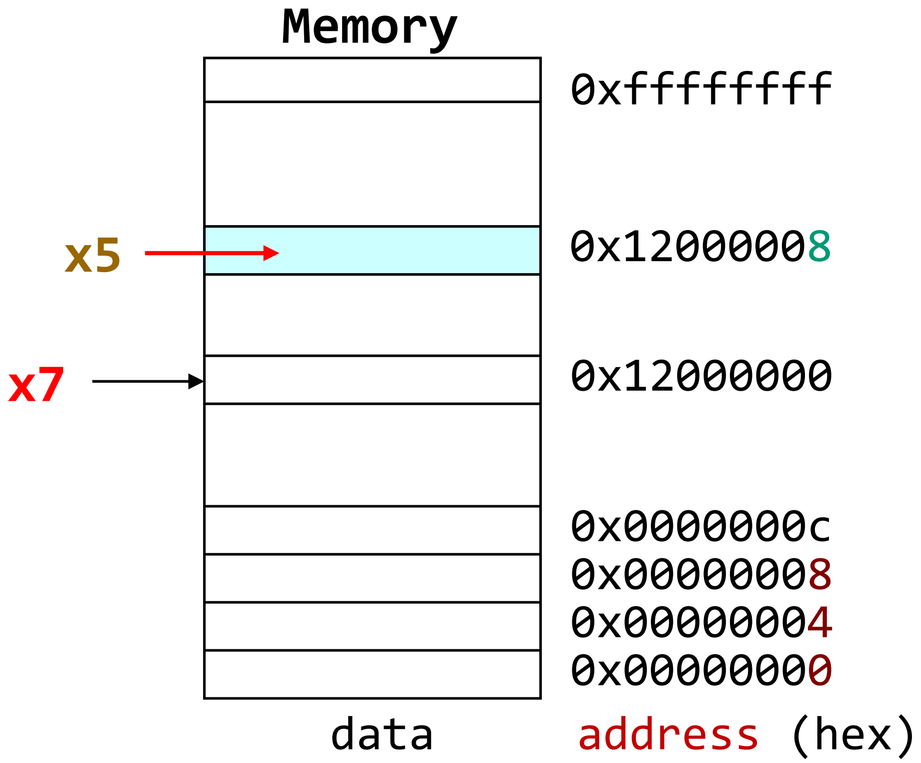
(4) Machine Language - Store word instruction (sw)

- Load Instruction Format (S-type):

```
sw x5, 8(x7)
```

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode

S-type



Exercise 3

- Compiling using load and store

```
A[1] = h + A[2];
```

- Assume variable h is associated with register s2 and base address of the array *A* is in s3. What is the RISC-V assembly code for the C program?

```
t0 = A[2];  # address is s3 + 8
t1 = s2 + t0;
A[1] = t1;  # address is s3 + 4
```

(5) RISC-V branch if not equal instructions (bne)

- RISC-V conditional branch instructions (bne, branch if not equal) :

```
 bne x4, x5, Lbl  # go to Lbl if x4!=x5
```

```
        if (i==j) h = i + j;
```

```
        bne x4, x5, Lbl1   # if (i!=j) goto Lbl1
        add x6, x4, x5     # h = i + j;
Lbl1:   ...
```

- Instruction Format (B-type):

imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode	B-type

- How is the branch destination address specified?

Exercise 4

- Compiling using add, addi, and bne

```
void main(){
    int i, sum=0;
    for(i=1; i!=10; i++) sum = sum + i;
}
```

- What is the RISC-V assembly code for the C program?

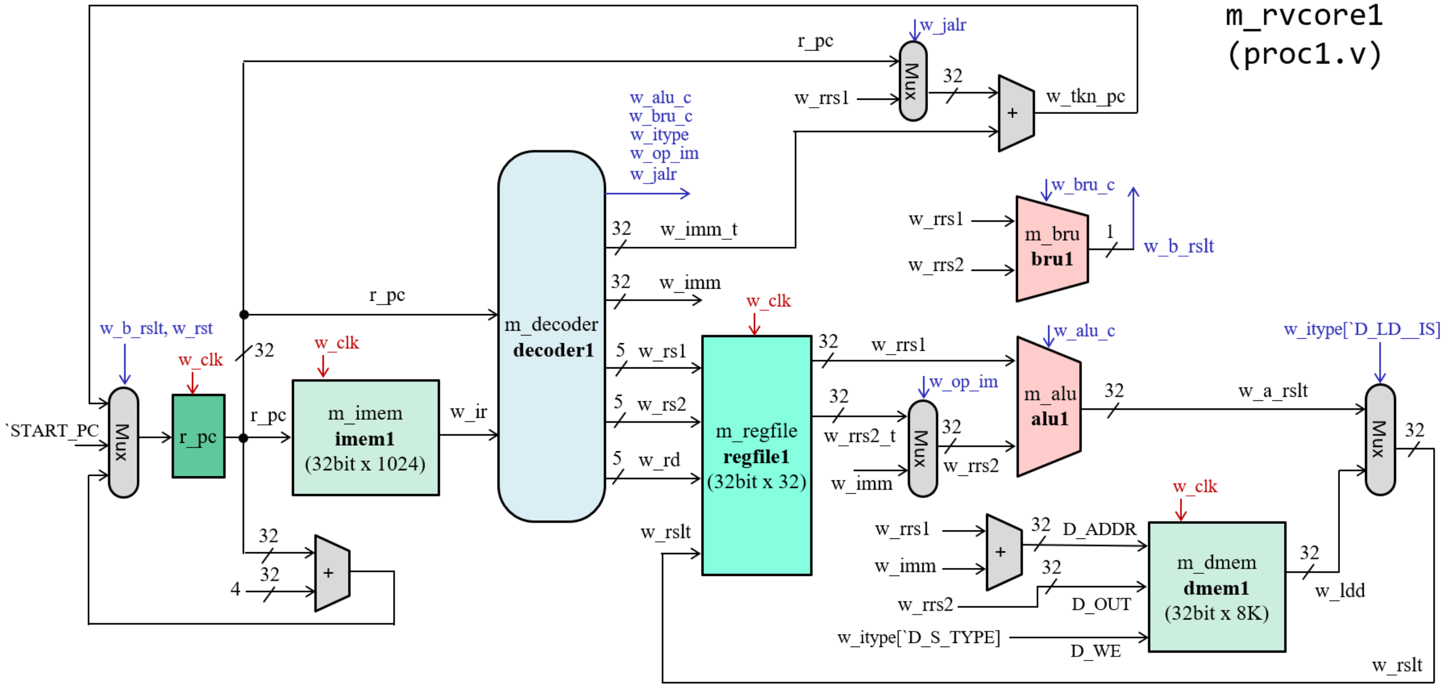
```
void main(){
    int t0, t1=10, t3=0;
    for(t0=1; t0!=t1; t0++) t3 = t3 + t0;
}
```

Typical five steps in processing an instruction

- IF: Instruction Fetch
  fetch an instruction from instruction memory or instruction cache
- ID: Instruction Decode
  decode an instruction and read input operands from register file
- EX: Execution
  perform operation, calculate an address of lw/sw
- MEM: Memory Access
  access data memory or data cache for lw/sw
- WB: Write Back
  write operation result and loaded data to register file

Single-cycle implementation of processors

- Single-cycle implementation also called single clock cycle implementation is the implementation in which an instruction is executed in one clock cycle.
  While easy to understand, it is too slow to be practical.
  It is useful as a baseline for lectures.

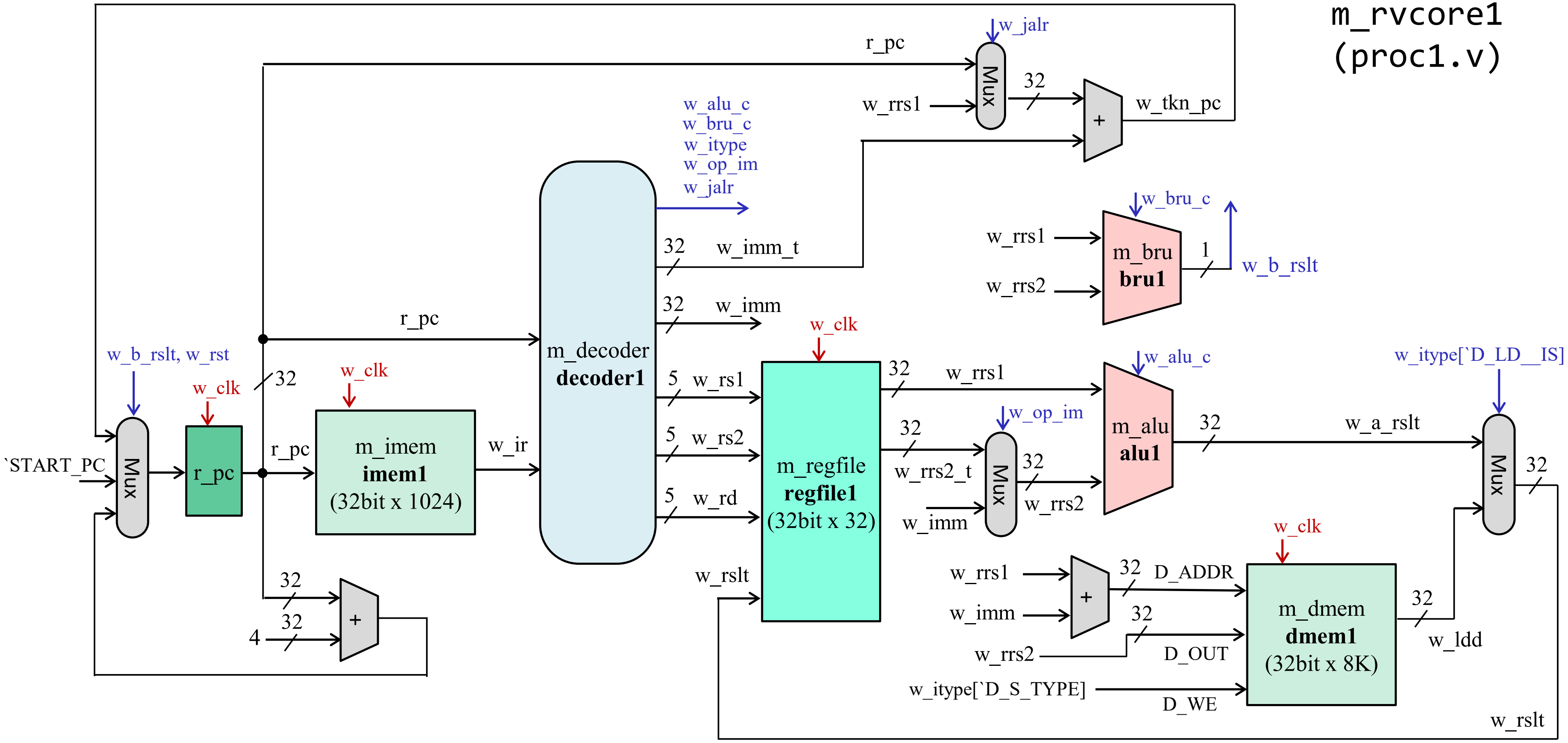


m_rvcore1
(proc1.v)

m_rvcore (RV32I, single-cycle processor)

- around 40MHz operating frequency for Arty A7 FPGA board
- lb, lbu, lh, lhu, sb, sh are not supported

m_rvcore1
(proc1.v)



Sample assembly code in RISC-V

- sample assembly code in the instruction memory
- the leftmost number is the instruction memory address where the instruction is stored
- the first register x0 is zero register with hardwiring 0

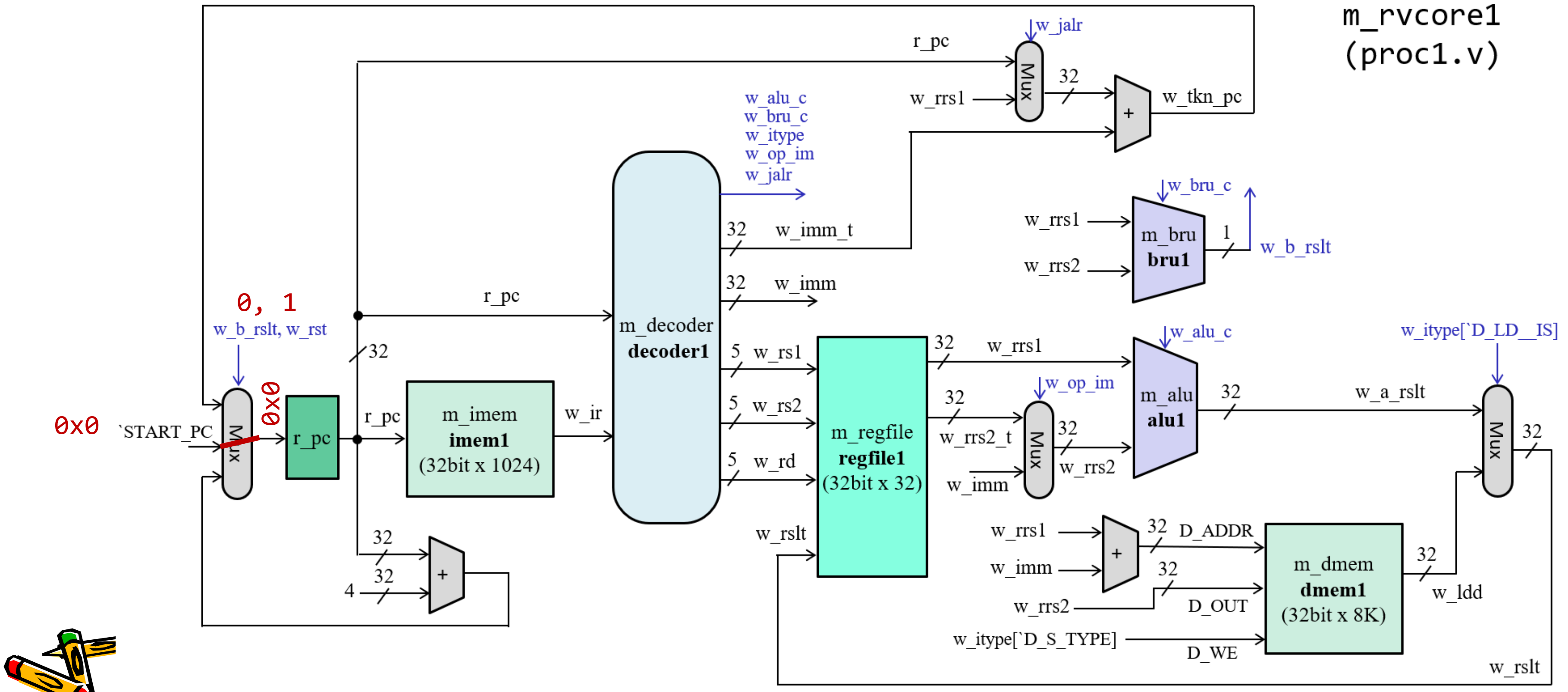
```
0x00  L1: addi x5, x0, 2       # x5 = 2
0x04      addi x6, x0, 3       # x6 = 3
0x08      add  x7, x5, x6      # x7 = x5 + x6 = 5
0x0c      sw   x7, 32(x0)      # mem[0 + 32] = x7 = 5
0x10      lw   x8, 32(x0)      # x8 = mem[0 + 32]
0x14      add  x9, x8, x5      # x9 = x8 + x5 = 7
0x18      bne  x5, x6, L1      # go to L1 if x5!=x6
```

```
0x00200293
0x00300313
0x006283B3
0x02702023
0x02002403
0x005404B3
0xFE6294E3
```

Processing behabior of rvcore1

```
0x00  L1: addi x5, x0, 2      # x5 = 2
0x04      addi x6, x0, 3      # x6 = 3
0x08      add  x7, x5, x6     # x7 = x5 + x6 = 5
0x0c      sw   x7, 32(x0)     # mem[0 + 32] = x7 = 5
0x10      lw   x8, 32(x0)     # x8 = mem[0 + 32]
0x14      add  x9, x8, x5     # x9 = x8 + x5 = 7
0x18      bne  x5, x6, L1     # go to L1 if x5!=x6
```

- cycle count 0 (cc0) at 50nsec

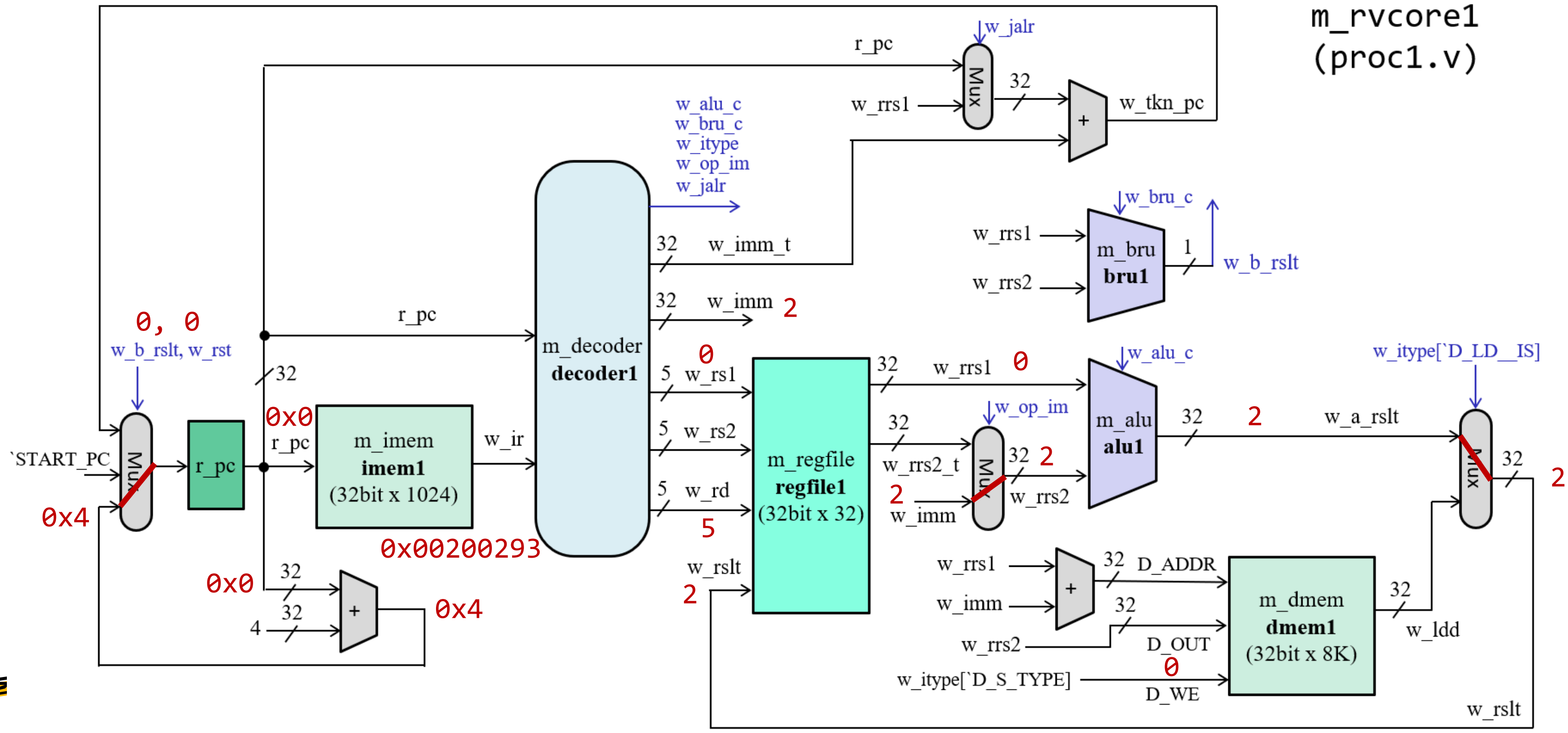


m_rvcore1
(proc1.v)

Processing behabior of rvcore1

```
0x00  L1: addi x5, x0, 2       # x5 = 2
0x04      addi x6, x0, 3       # x6 = 3
0x08      add  x7, x5, x6      # x7 = x5 + x6 = 5
0x0c      sw   x7, 32(x0)      # mem[0 + 32] = x7 = 5
0x10      lw   x8, 32(x0)      # x8 = mem[0 + 32]
0x14      add  x9, x8, x5      # x9 = x8 + x5 = 7
0x18      bne  x5, x6, L1      # go to L1 if x5!=x6
```

- cycle count 1 (cc1) at 150nsec
- executing
  `addi x5, x0, 2`
  of address 0x00

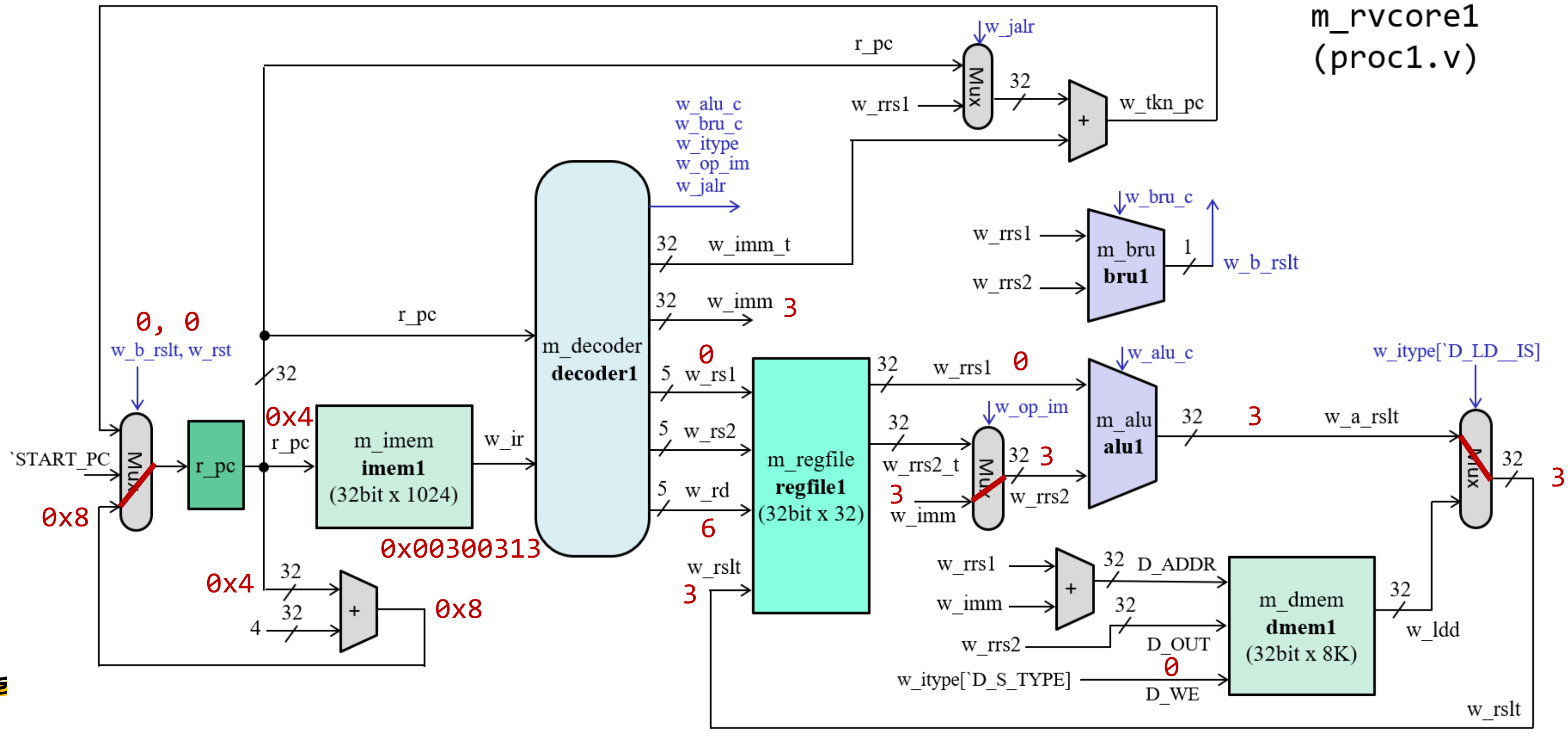


m_rvcore1
(proc1.v)

Processing behabior of rvcore1

```
0x00  L1: addi x5, x0, 2       # x5 = 2
0x04      addi x6, x0, 3       # x6 = 3
0x08      add  x7, x5, x6      # x7 = x5 + x6 = 5
0x0c      sw   x7, 32(x0)      # mem[0 + 32] = x7 = 5
0x10      lw   x8, 32(x0)      # x8 = mem[0 + 32]
0x14      add  x9, x8, x5      # x9 = x8 + x5 = 7
0x18      bne  x5, x6, L1      # go to L1 if x5!=x6
```

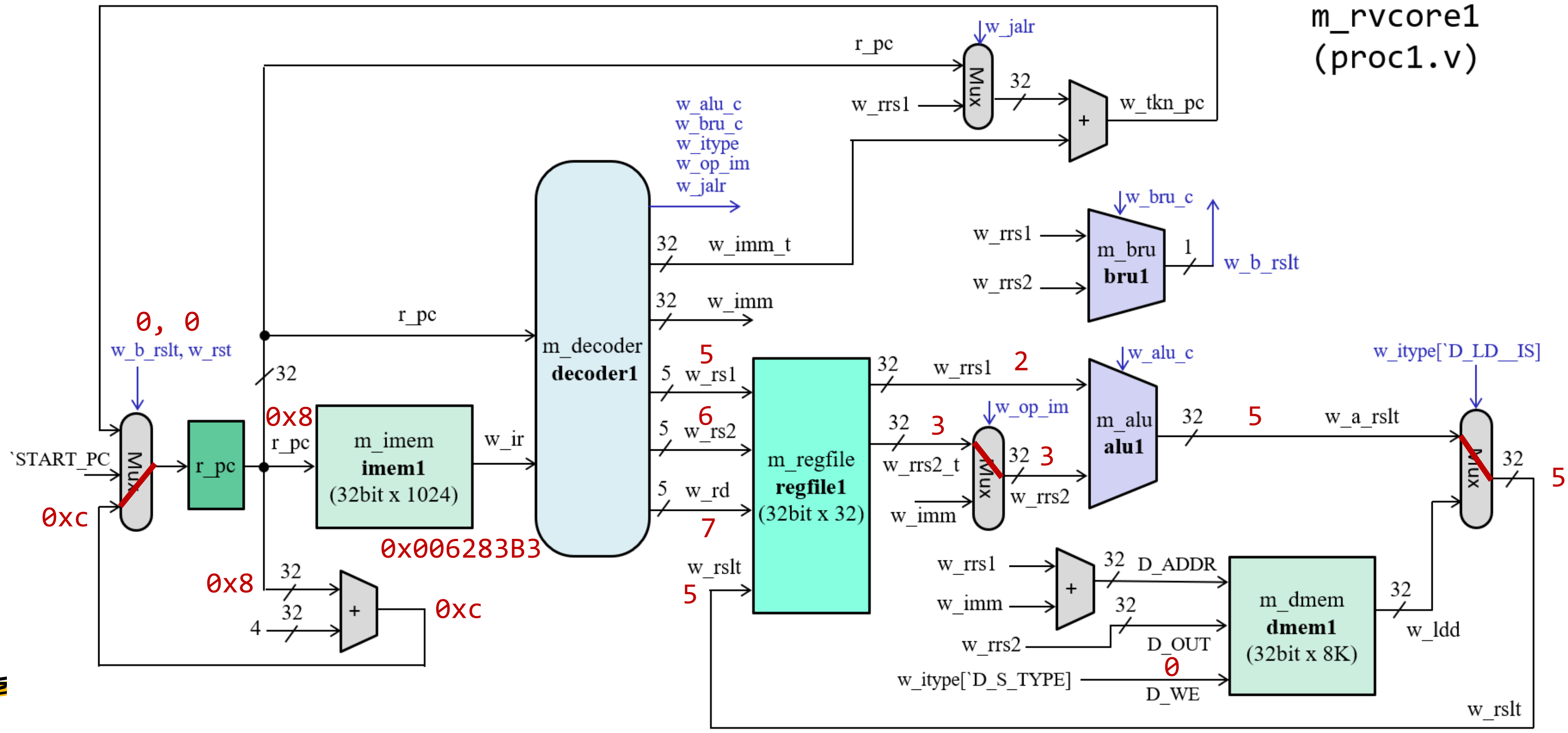
- cycle count 2 (cc2) at 250nsec
- executing `addi x6, x0, 3` of address 0x04



Processing behabior of rvcore1

```
0x00  L1: addi x5, x0, 2       # x5 = 2
0x04      addi x6, x0, 3       # x6 = 3
0x08      add  x7, x5, x6      # x7 = x5 + x6 = 5
0x0c      sw   x7, 32(x0)      # mem[0 + 32] = x7 = 5
0x10      lw   x8, 32(x0)      # x8 = mem[0 + 32]
0x14      add  x9, x8, x5      # x9 = x8 + x5 = 7
0x18      bne  x5, x6, L1      # go to L1 if x5!=x6
```

- cycle count 3 (cc3) at 350nsec
- executing
  `add  x7, x5, x6`
  of address 0x08

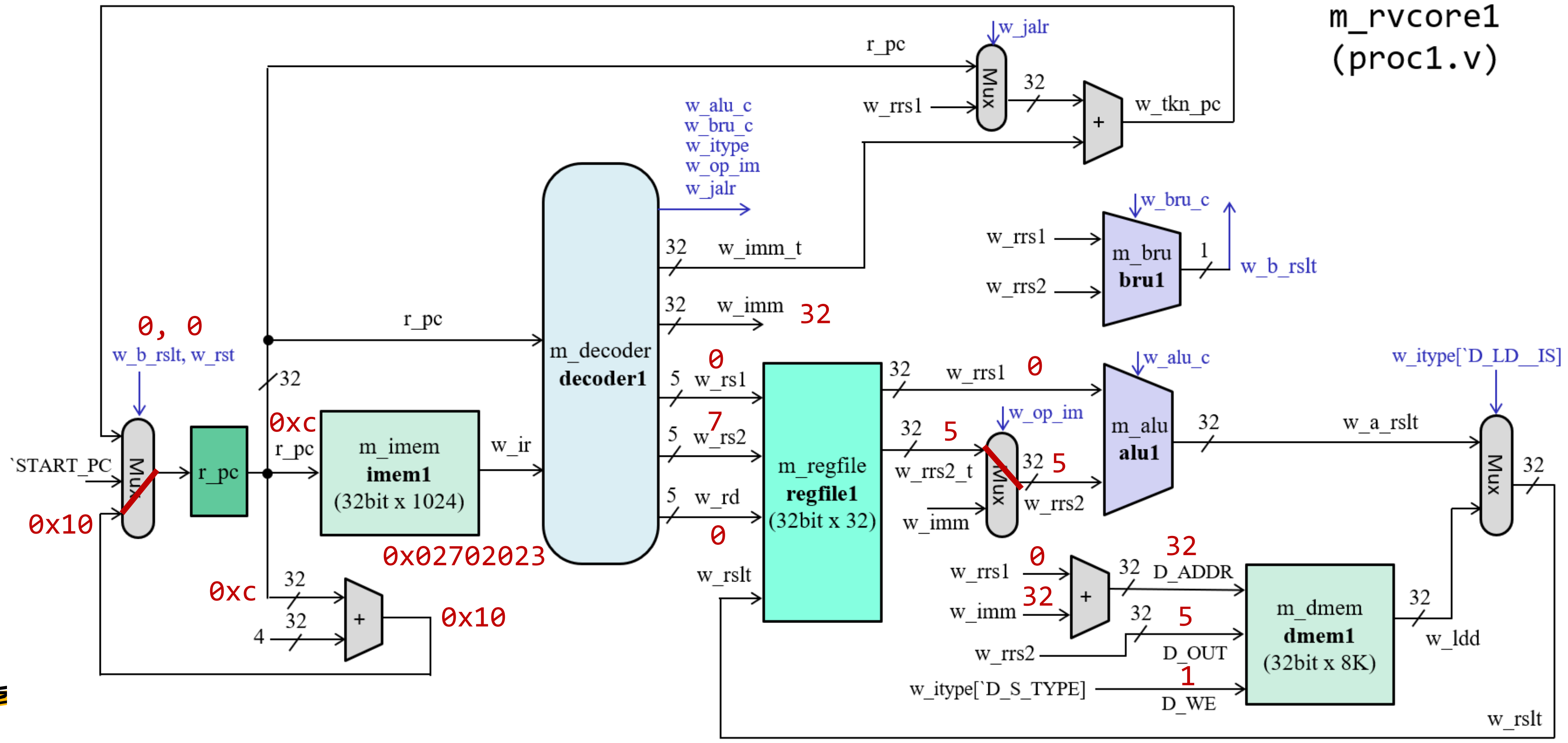


Processing behabior of rvcore1

```
0x00  L1: addi x5, x0, 2       # x5 = 2
0x04      addi x6, x0, 3       # x6 = 3
0x08      add  x7, x5, x6      # x7 = x5 + x6 = 5
0x0c      sw   x7, 32(x0)      # mem[0 + 32] = x7 = 5
0x10      lw   x8, 32(x0)      # x8 = mem[0 + 32]
0x14      add  x9, x8, x5      # x9 = x8 + x5 = 7
0x18      bne  x5, x6, L1      # go to L1 if x5!=x6
```

- cycle count 4 (cc4) at 450nsec
- executing `sw   x7, 32(x0)` of address 0x0c

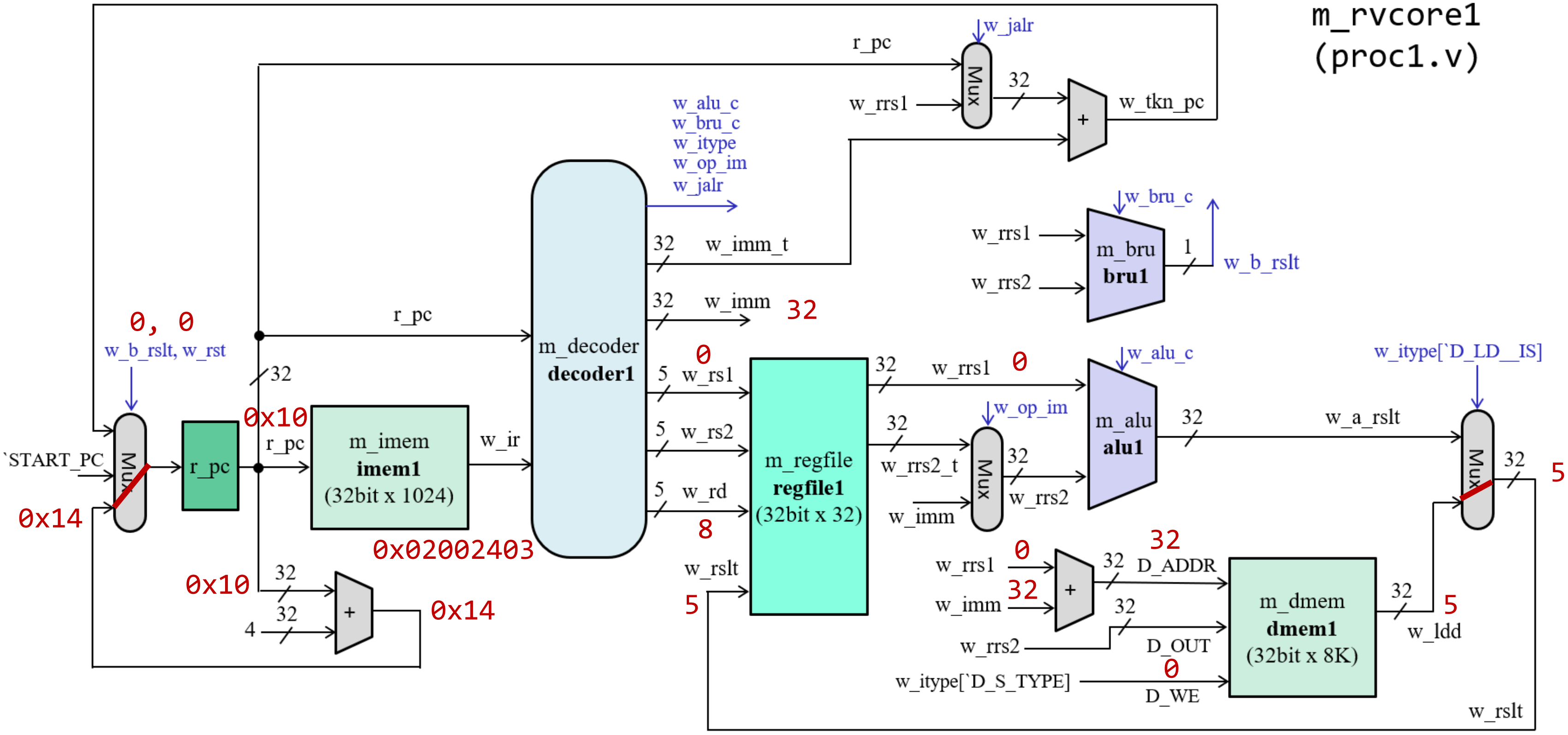
m_rvcore1 (proc1.v)



Processing behabior of rvcore1

```
0x00  L1: addi x5, x0, 2       # x5 = 2
0x04      addi x6, x0, 3       # x6 = 3
0x08      add  x7, x5, x6      # x7 = x5 + x6 = 5
0x0c      sw   x7, 32(x0)      # mem[0 + 32] = x7 = 5
0x10      lw   x8, 32(x0)      # x8 = mem[0 + 32]
0x14      add  x9, x8, x5      # x9 = x8 + x5 = 7
0x18      bne  x5, x6, L1      # go to L1 if x5!=x6
```

- cycle count 5 (cc5) at 550nsec
- executing
  `lw   x8, 32(x0)`
  of address 0x10

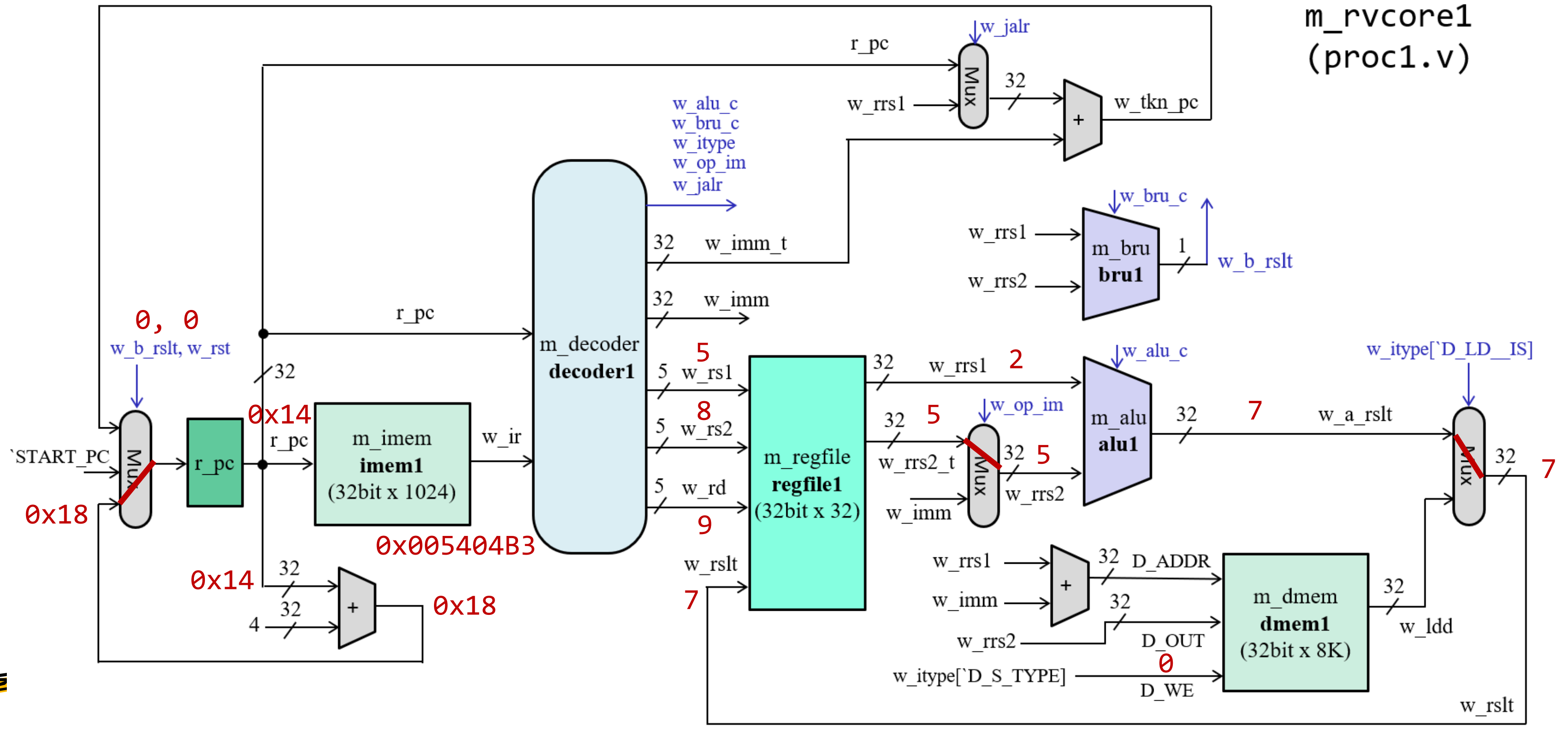


m_rvcore1
(proc1.v)

Processing behabior of rvcore1

```
0x00  L1: addi x5, x0, 2       # x5 = 2
0x04      addi x6, x0, 3       # x6 = 3
0x08      add  x7, x5, x6      # x7 = x5 + x6 = 5
0x0c      sw   x7, 32(x0)      # mem[0 + 32] = x7 = 5
0x10      lw   x8, 32(x0)      # x8 = mem[0 + 32]
0x14      add  x9, x8, x5      # x9 = x8 + x5 = 7
0x18      bne  x5, x6, L1      # go to L1 if x5!=x6
```

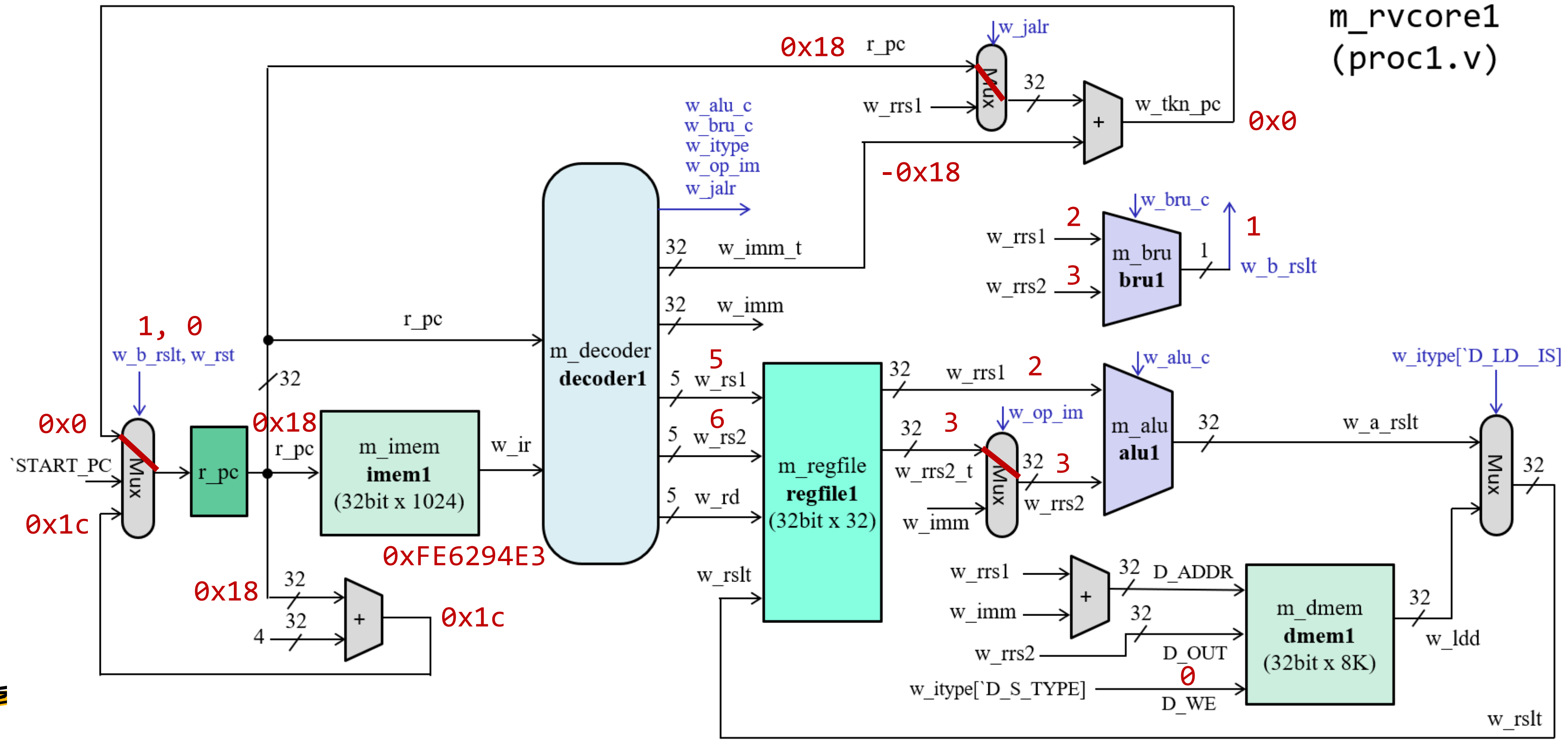
- cycle count 6 (cc6) at 650nsec
- executing `add  x9, x8, x5` of address 0x14



Processing behabior of rvcore1

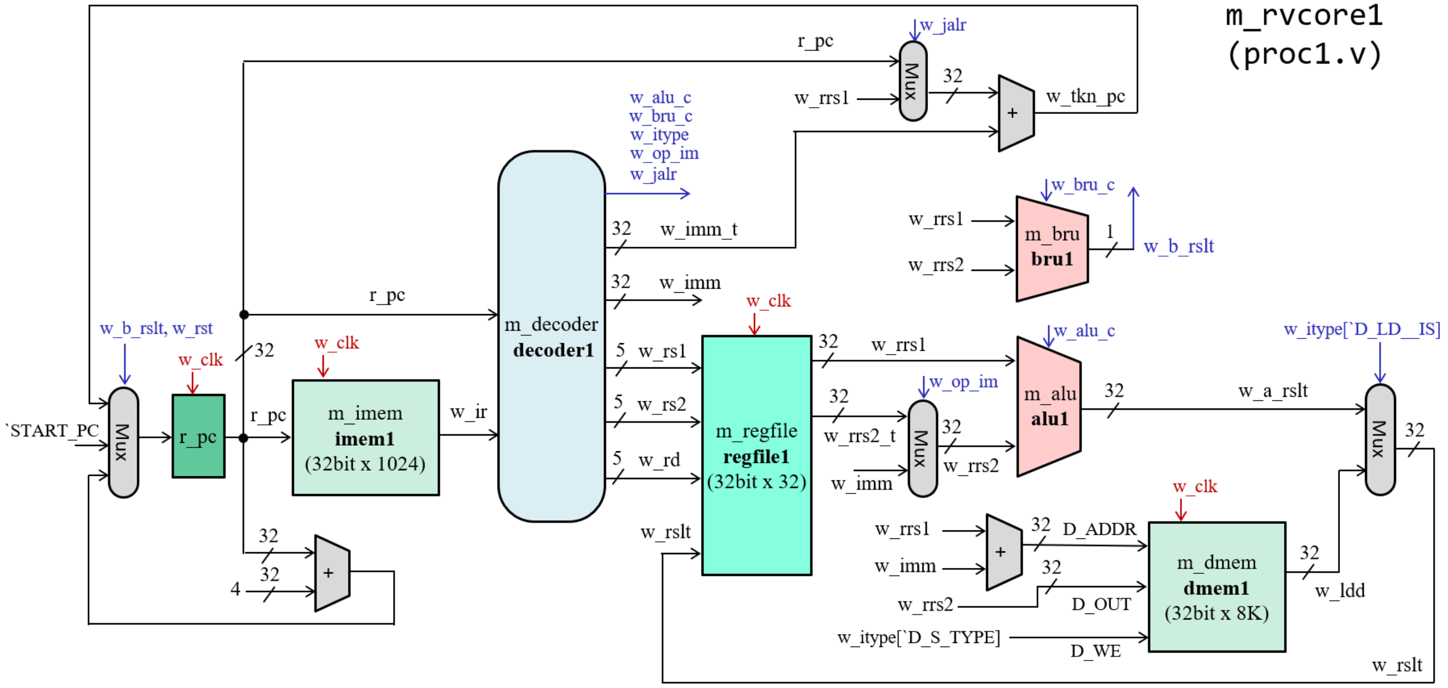
```
0x00  L1: addi x5, x0, 2      # x5 = 2
0x04      addi x6, x0, 3      # x6 = 3
0x08      add  x7, x5, x6     # x7 = x5 + x6 = 5
0x0c      sw   x7, 32(x0)     # mem[0 + 32] = x7 = 5
0x10      lw   x8, 32(x0)     # x8 = mem[0 + 32]
0x14      add  x9, x8, x5     # x9 = x8 + x5 = 7
0x18      bne  x5, x6, L1     # go to L1 if x5!=x6
```

- cycle count 7 (cc7) at 750nsec
- executing `bne   x5, x6, L1` of address 0x18



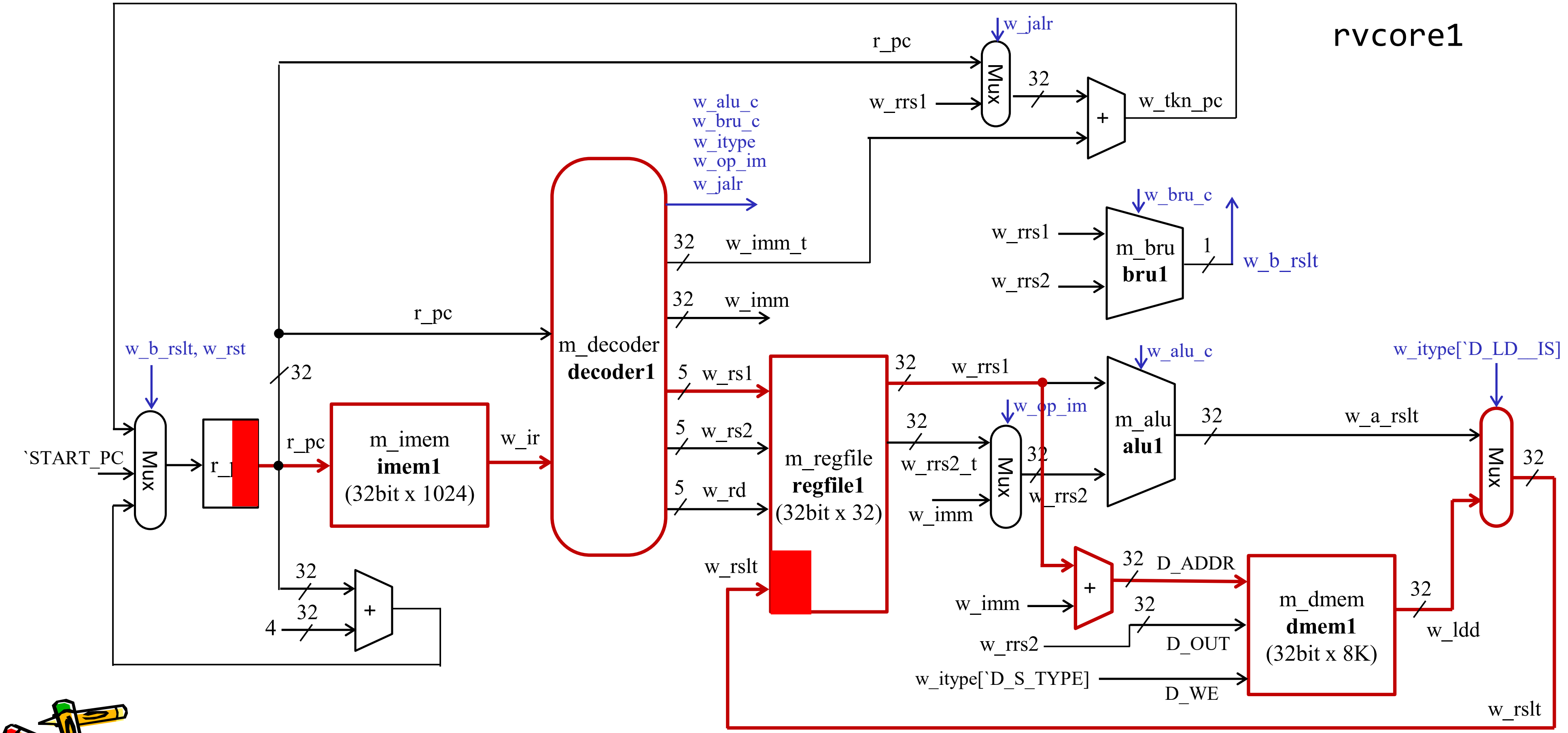
Single-cycle implementation of processors

- Single-cycle implementation also called single clock cycle implementation is the implementation in which an instruction is executed in one clock cycle.
  While easy to understand, it is too slow to be practical.
  It is useful as a baseline for lectures.



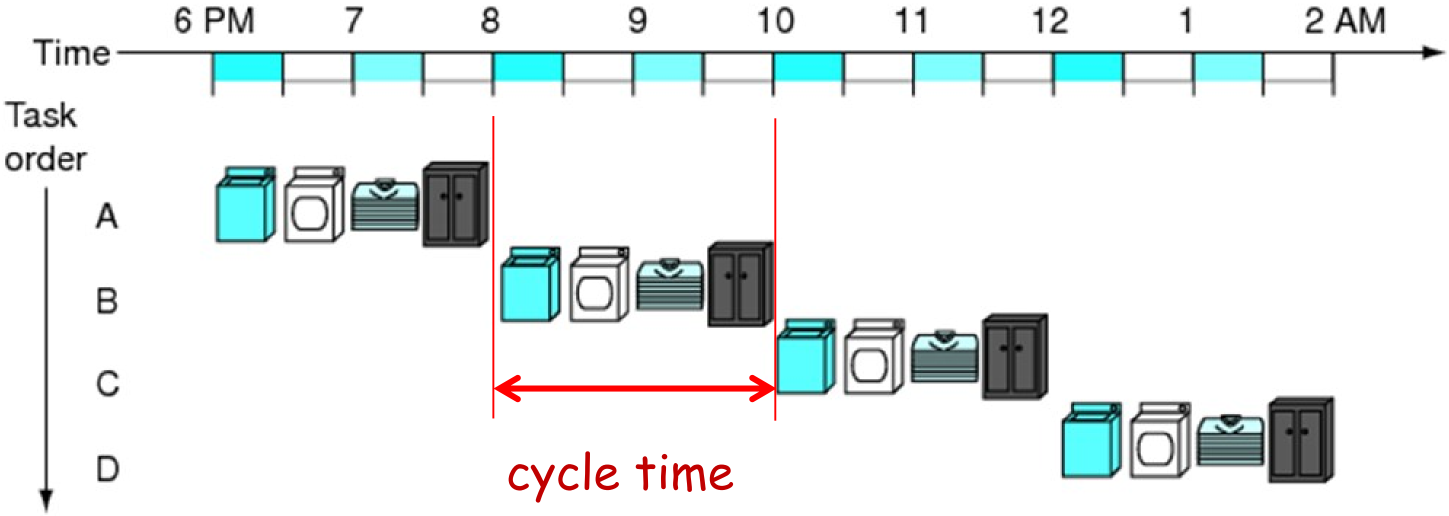
Critical path of rvcore1 (single-cycle version)

- The critical path is defined as the path between a source register (or memory) and a destination register with the maximum delay.
- This design is too slow to be practical.



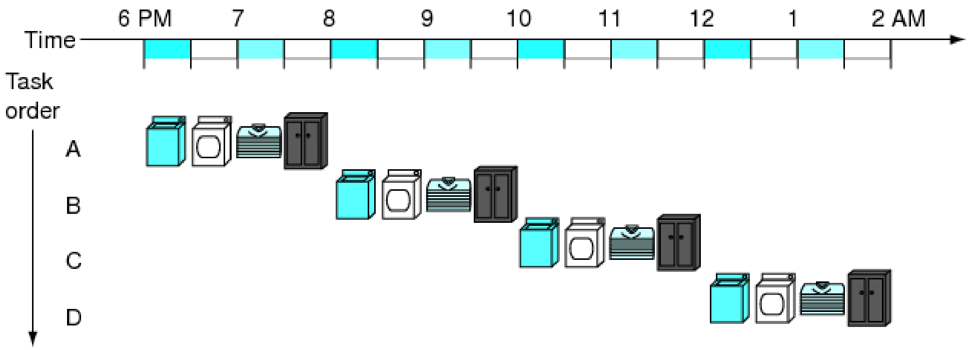
Single-cycle implementation of laundry

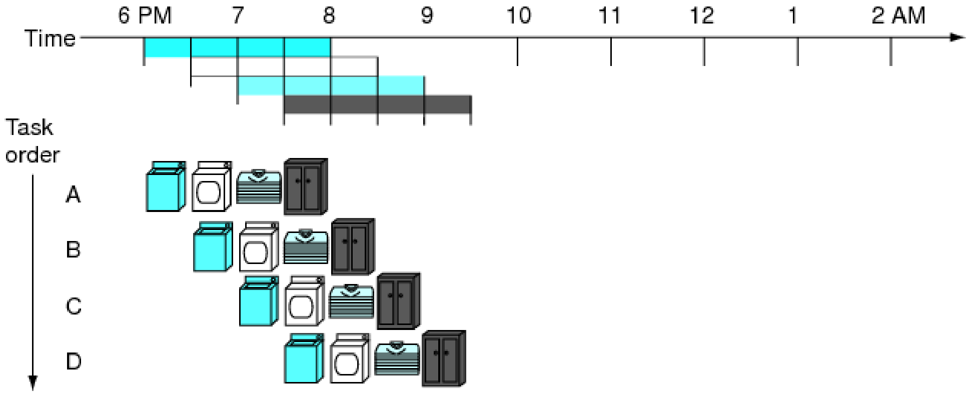
- (A) Ann, (B) Brian, (C) Cathy, and (D) Don each have dirty clothes to be *washed*, *dried*, *folded*, and *put away*, each taking 30 minutes.
- The cycle time (the time from the end of one load to the end of the next one) is 2 hours.
- For four loads, the sequential laundry takes 8 hours.



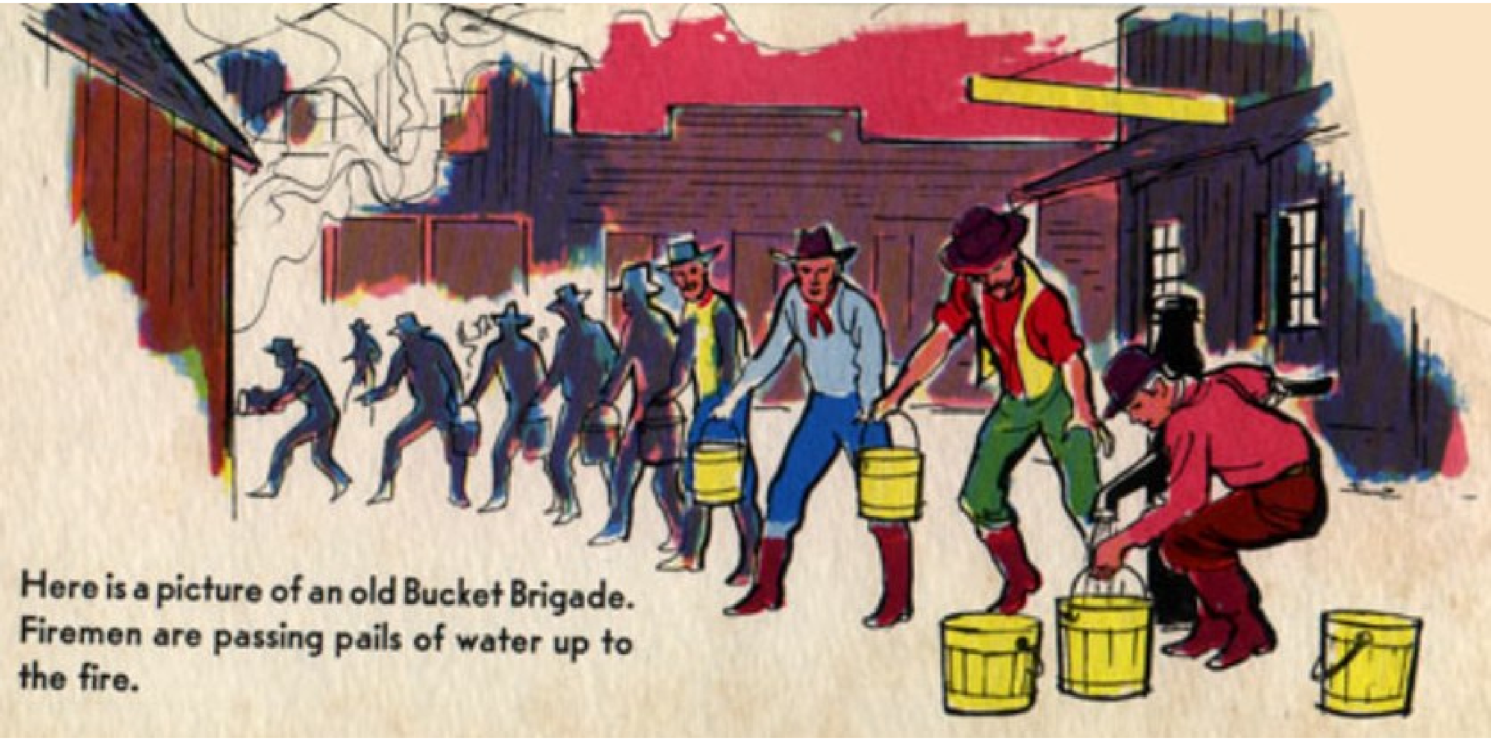
Single-cycle implementation and pipelining

- When the washing of load A is finished at 6:30 p.m., another washing of load B starts.
- Pipelined laundry takes 3.5 hours just using the same hardware resources. The cycle time is 30 minutes.
- What is the latency (execution time) of each load?





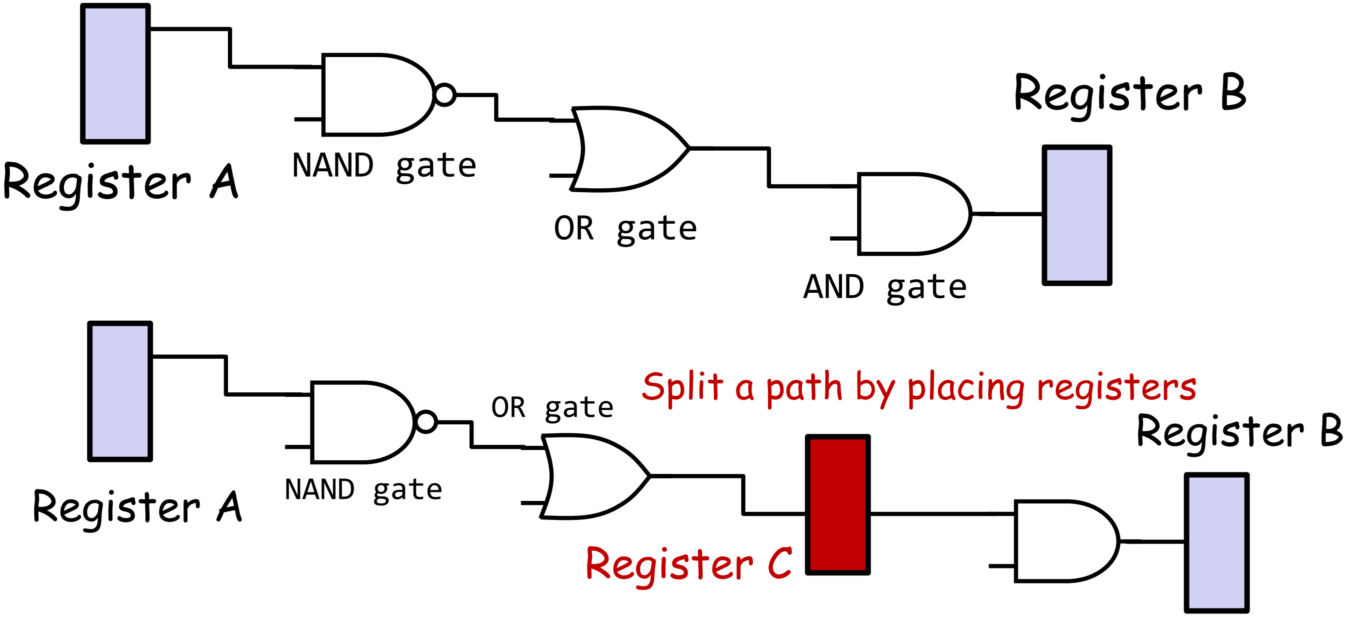
Bucket brigade



Here is a picture of an old Bucket Brigade. Firemen are passing pails of water up to the fire.

Clock rate is mainly determined by

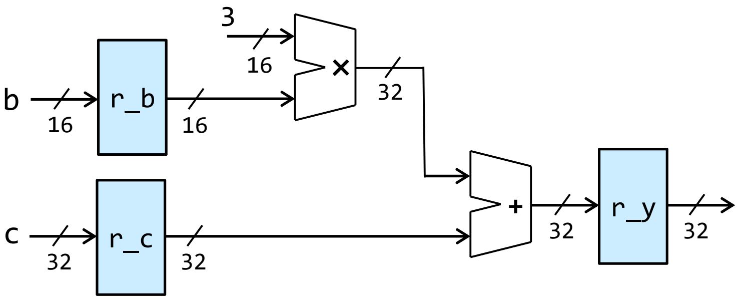
- Switching speed of gates (transistors)
- The number of levels of gates
  - The maximum number of gates cascaded in series in any combinational logics.
  - In this example, the number of levels of gates is 3.
- Wiring delay and fanout

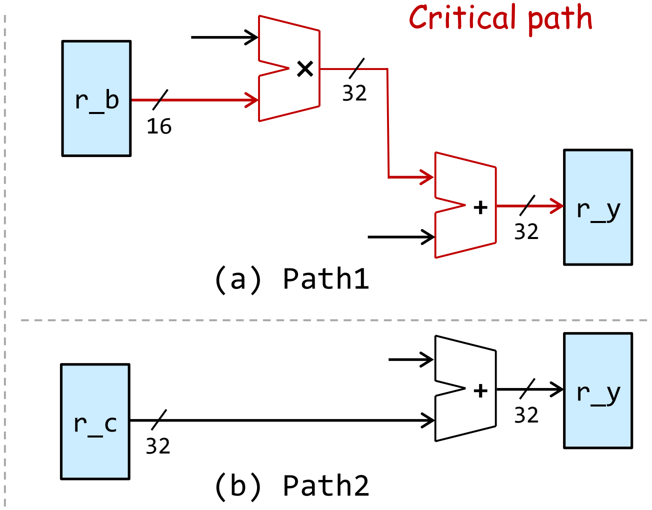


Split a path by placing registers

Pipelining example: multiply-add operation (1)

- As an example of pipelining, we will see a multiply-add circuit.
- r_b, r_c are input registers and r_y is output register of the circuit.
- This has two paths named path1 and path2, and path1 is the critical path to determine the maximum operating frequency.



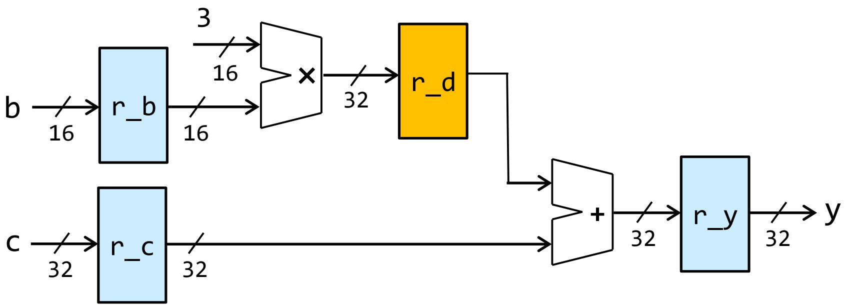


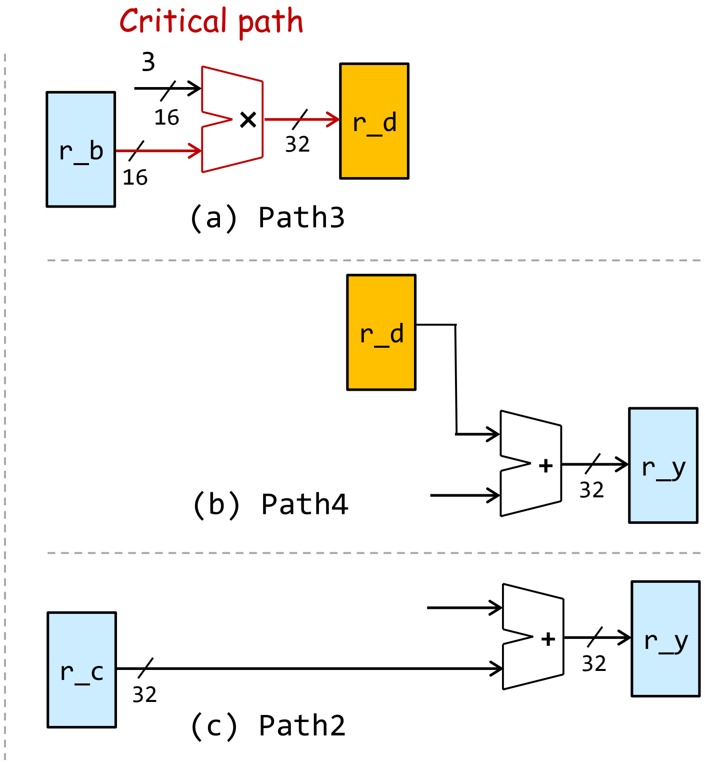
(a) Path1

(b) Path2

Pipelining example: multiply-add operation (2)

- By inserting register r_d, the critical path can be divided into Path3 and Path4.
- As a result, the new critical path becomes Path3.
- This has the disadvantage that input b and c in the same clock cycle cannot be processed.





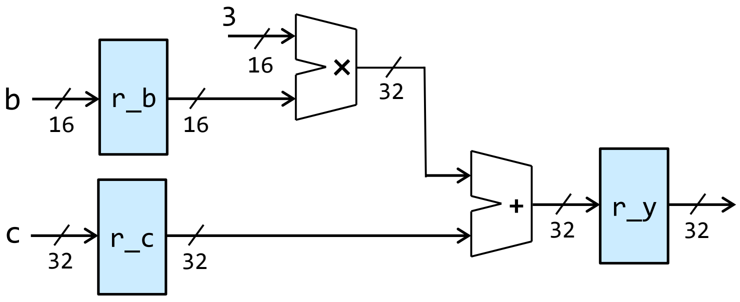
(a) Path3

(b) Path4

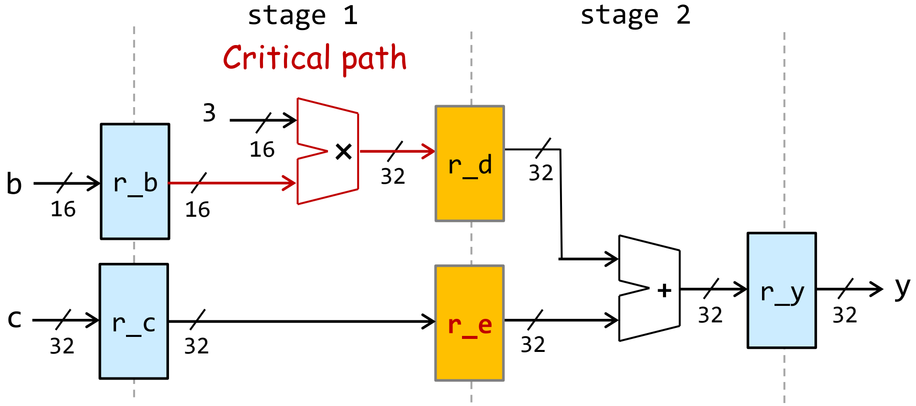
(c) Path2

Pipelining example: multiply-add operation (3)

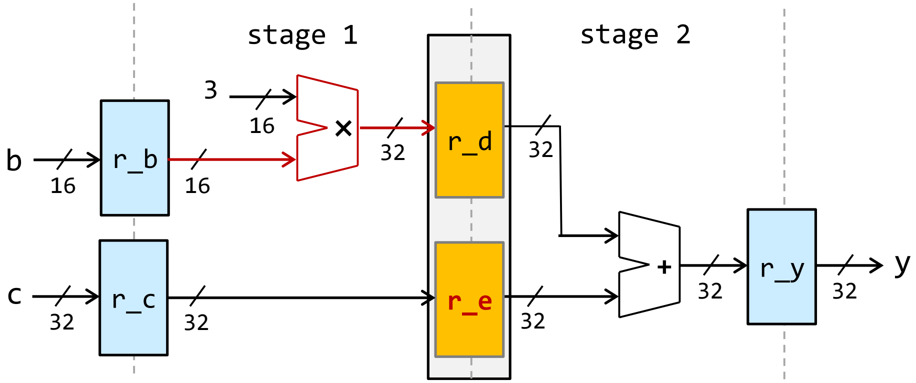
- To overcome this drawback, we insert register r_e.
- This realizes a pipeline with stages 1 and 2.
A set of registers between two adjacent stages are called a pipeline register.



(a) original multiply-add circuit



(b) two-stage pipelined circuit



pipeline register