CS 271
Computer Architecture & Assembly Language

Lecture 2

Intro to IA-32 and MASM

1/6/22, Thursday

Oregon State University

Due Reminder

- Week 1 Summary Exercise:
  - Due Sunday 1/9 11:59 pm on Canvas

Lecture Topics:

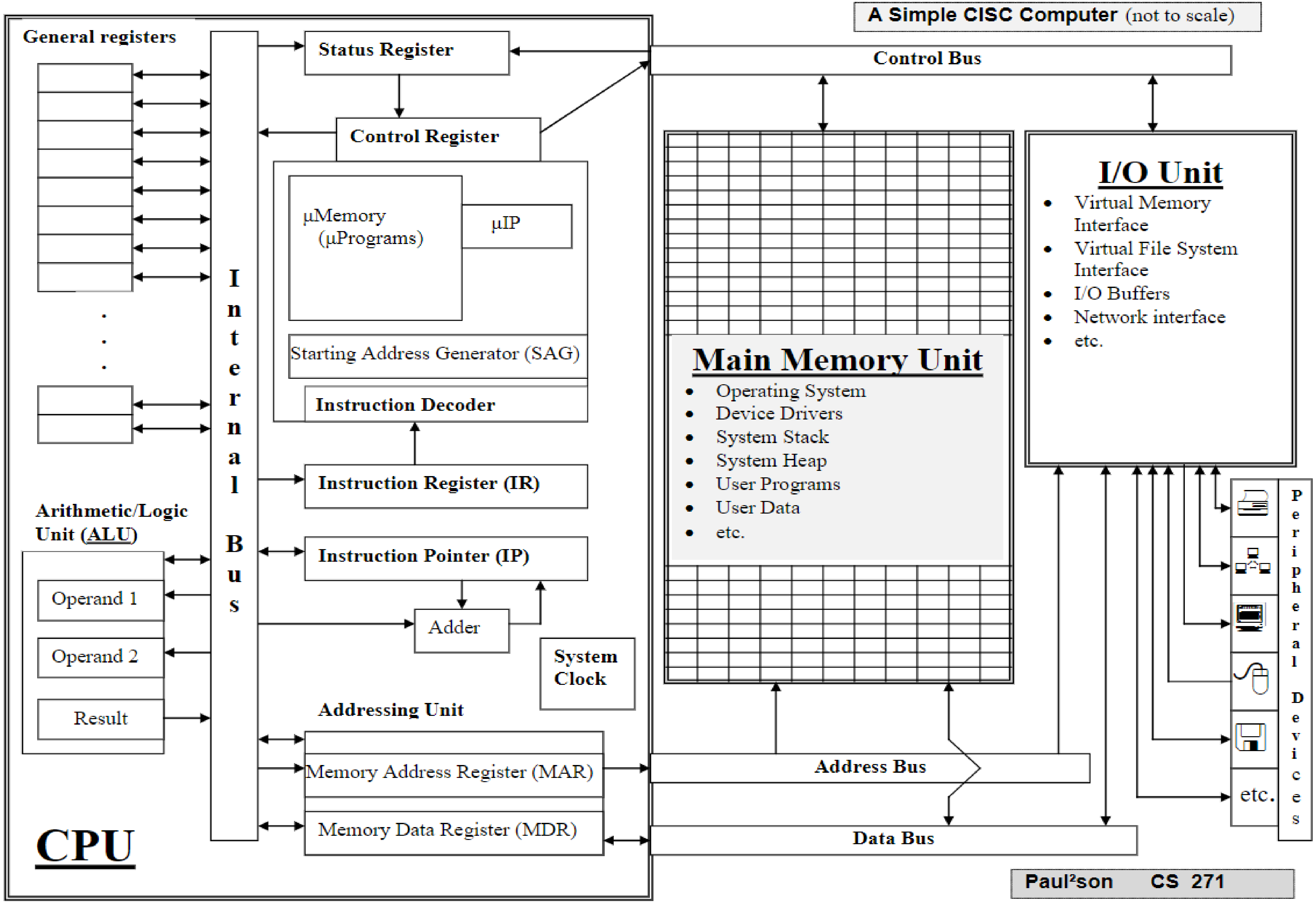
- How computer hardware works?
- Introduction to Intel IA-32 architecture
- Introduction to MASM assembly language
- Writing a MASM program

Preliminaries

- Inside a computer, information is represented electrically
  - Smallest unit of information is a switch (may be on or off)

- We often represent “off” as 0 and “on” as 1, so a single switch represents a binary digit, and is called a “**bit**”.

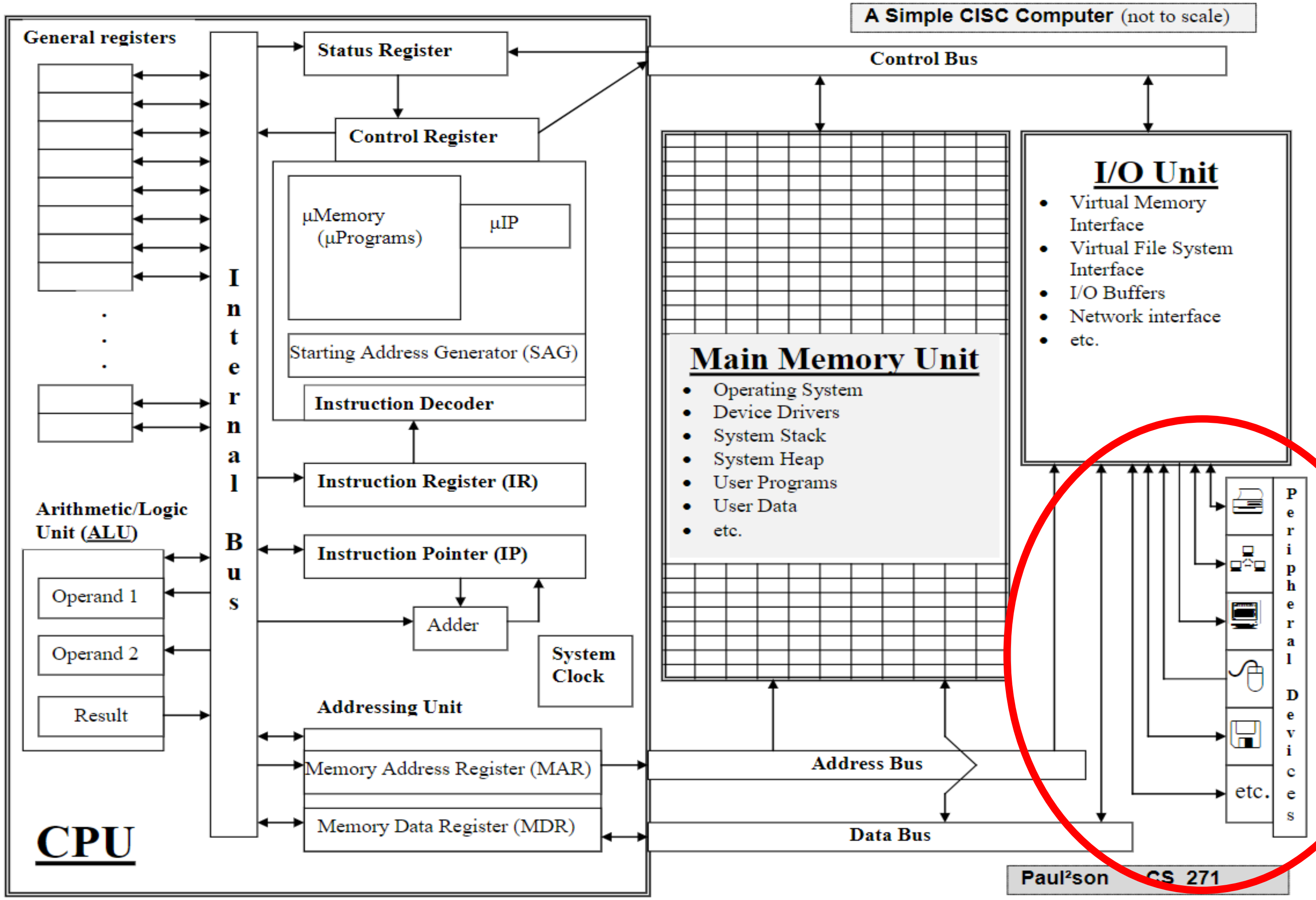
- Different combinations of switches represent different information
  - A group of 8 bits is called a **byte**



A Simple CISC Computer (not to scale)
General registers
Status Register
Control Register
Control Bus
μMemory
(μPrograms)
μIP
Starting Address Generator (SAG)
Instruction Decoder
Internal Bus
Instruction Register (IR)
Instruction Pointer (IP)
Adder
System Clock
Arithmetic/Logic Unit (ALU)
Operand 1
Operand 2
Result
Addressing Unit
Memory Address Register (MAR)
Memory Data Register (MDR)
CPU
Main Memory Unit
Operating System
Device Drivers
System Stack
System Heap
User Programs
User Data
etc.
I/O Unit
Virtual Memory Interface
Virtual File System Interface
I/O Buffers
Network interface
etc.
Peripheral Devices
etc.
Address Bus
Data Bus
Paul²son CS 271

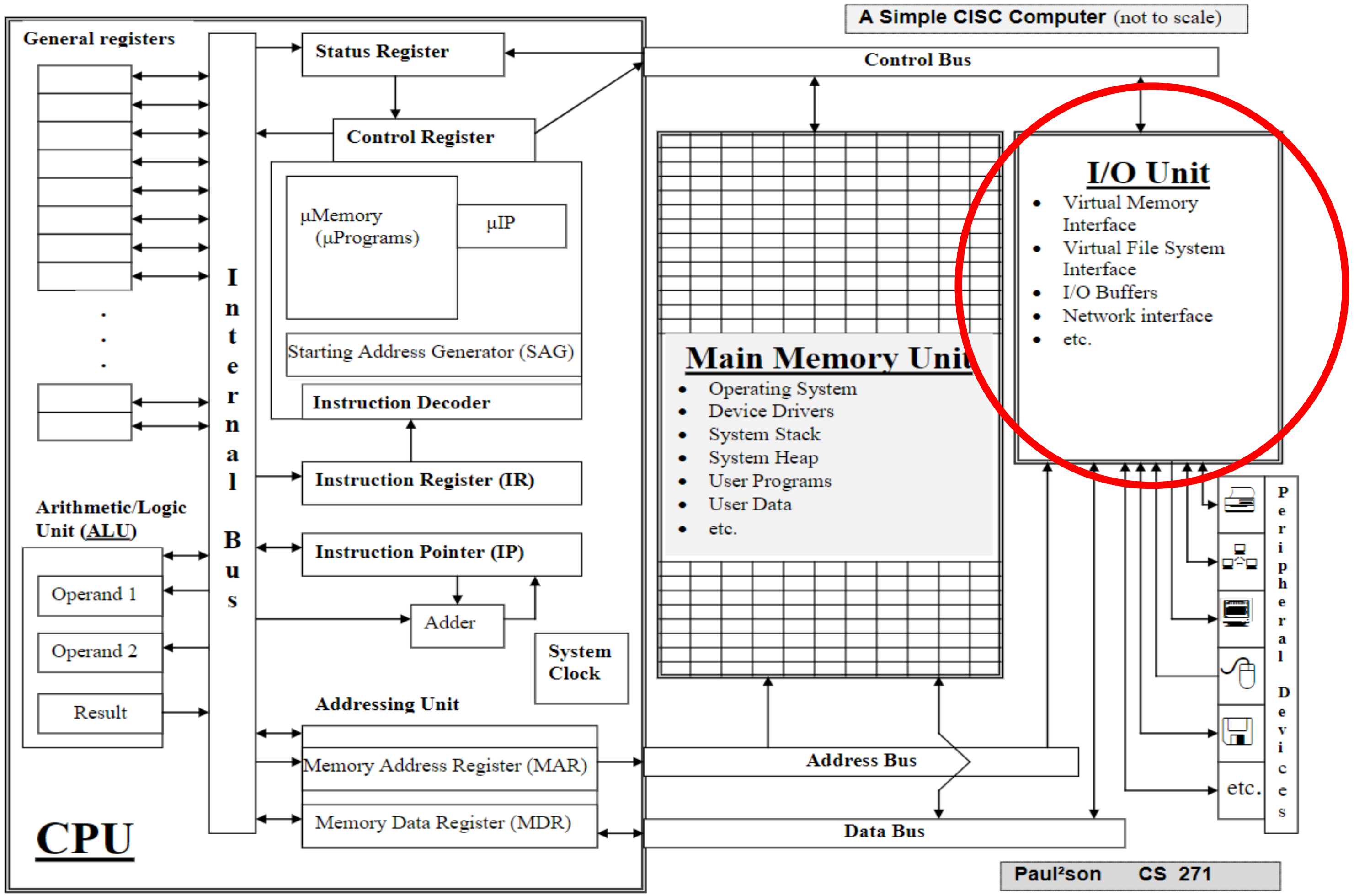
Main parts of the CISC diagram (Complex Instruction Set Computer)

- **Peripheral Devices**: External devices:
  - Store/retrieve data (non-volatile storage)
  - Convert data between human-readable and machine-readable forms



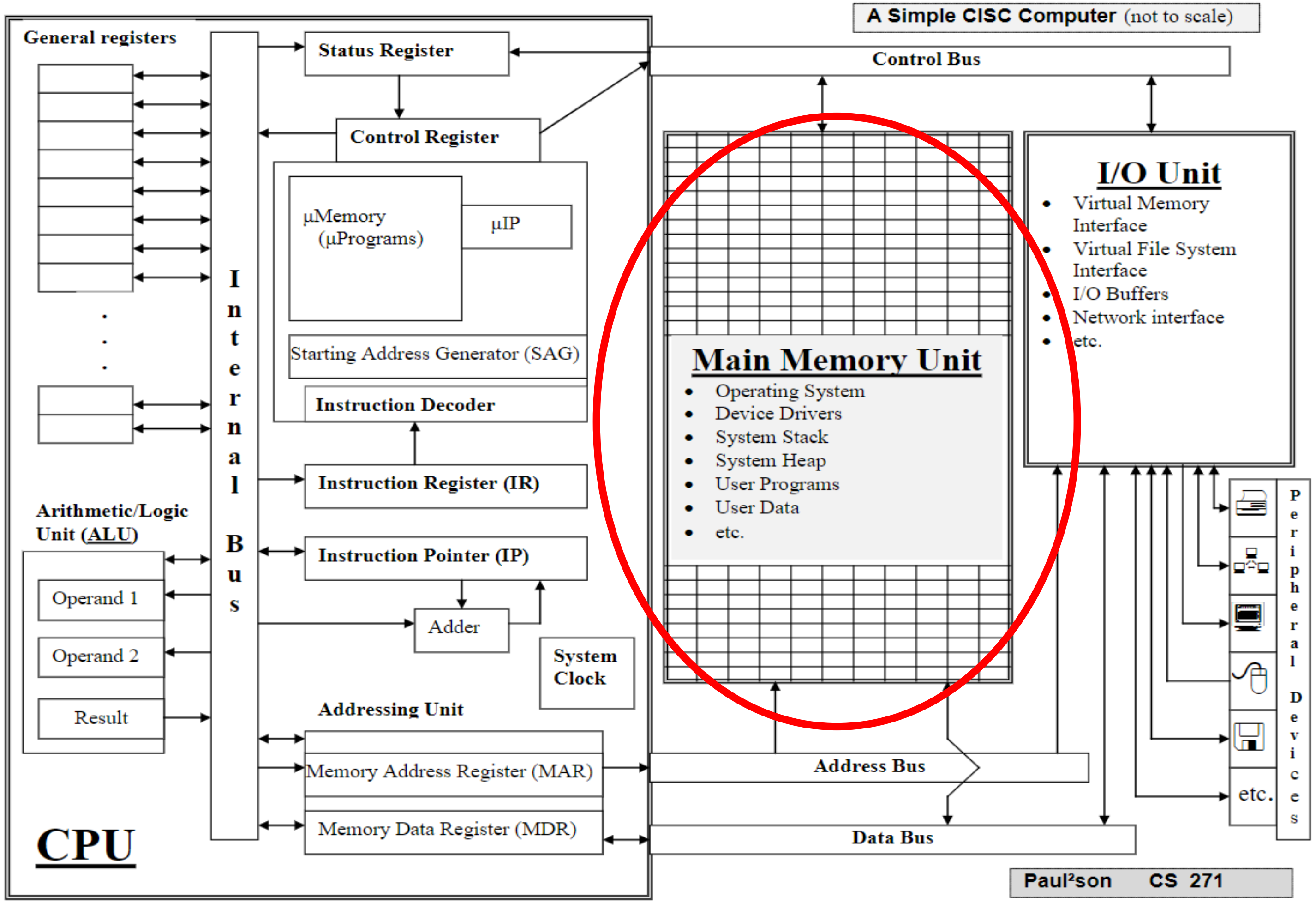
Main parts of the CISC diagram (Complex Instruction Set Computer)

- **I/O Unit**: Hardware/software functions:
  - Communicate between CPU/Memory and peripheral devices



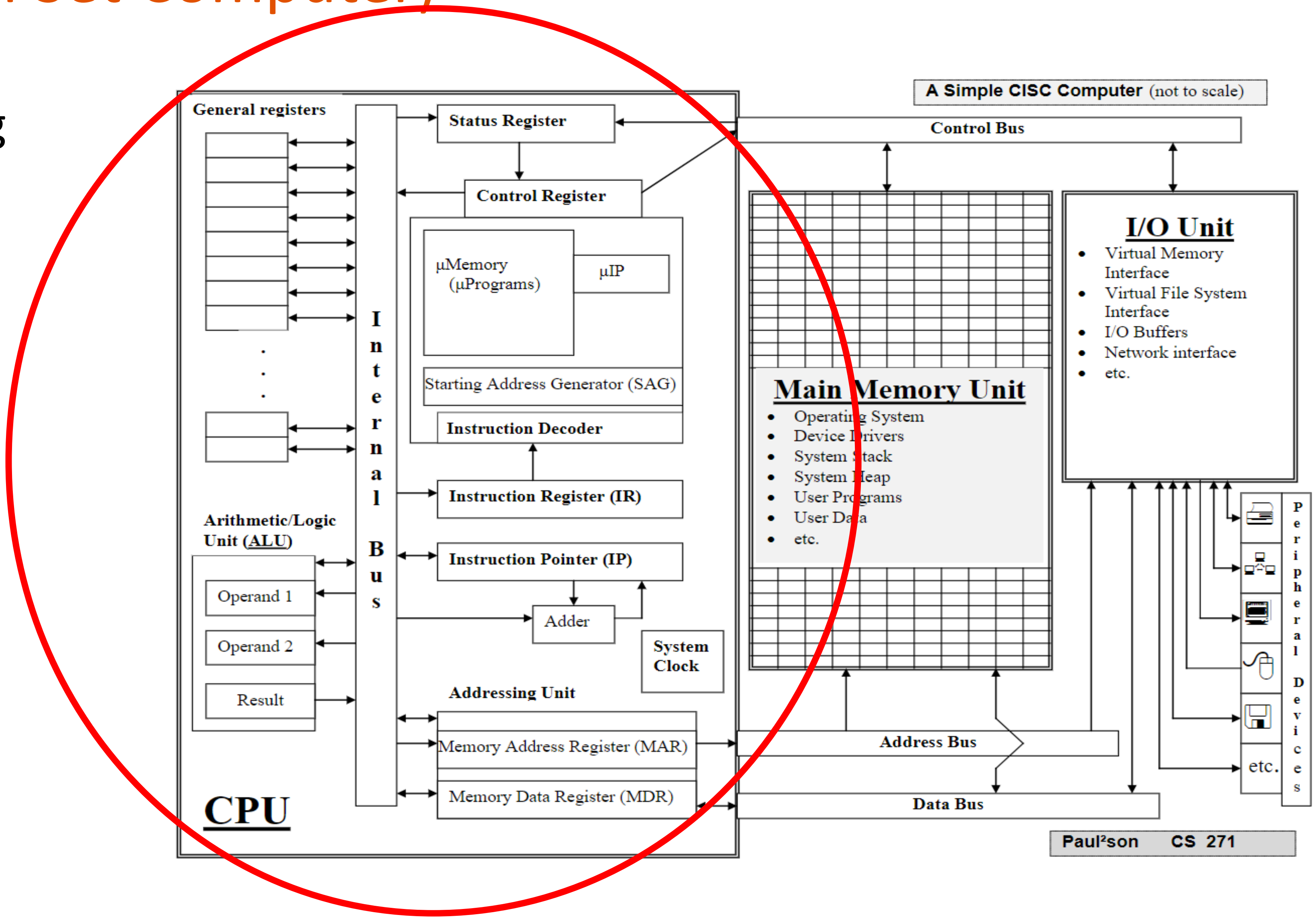
Main parts of the CISC diagram (Complex Instruction Set Computer)

- **Main Memory Unit**: Cells with addresses:
  - Store programs and data currently being used by the CPU (volatile storage)



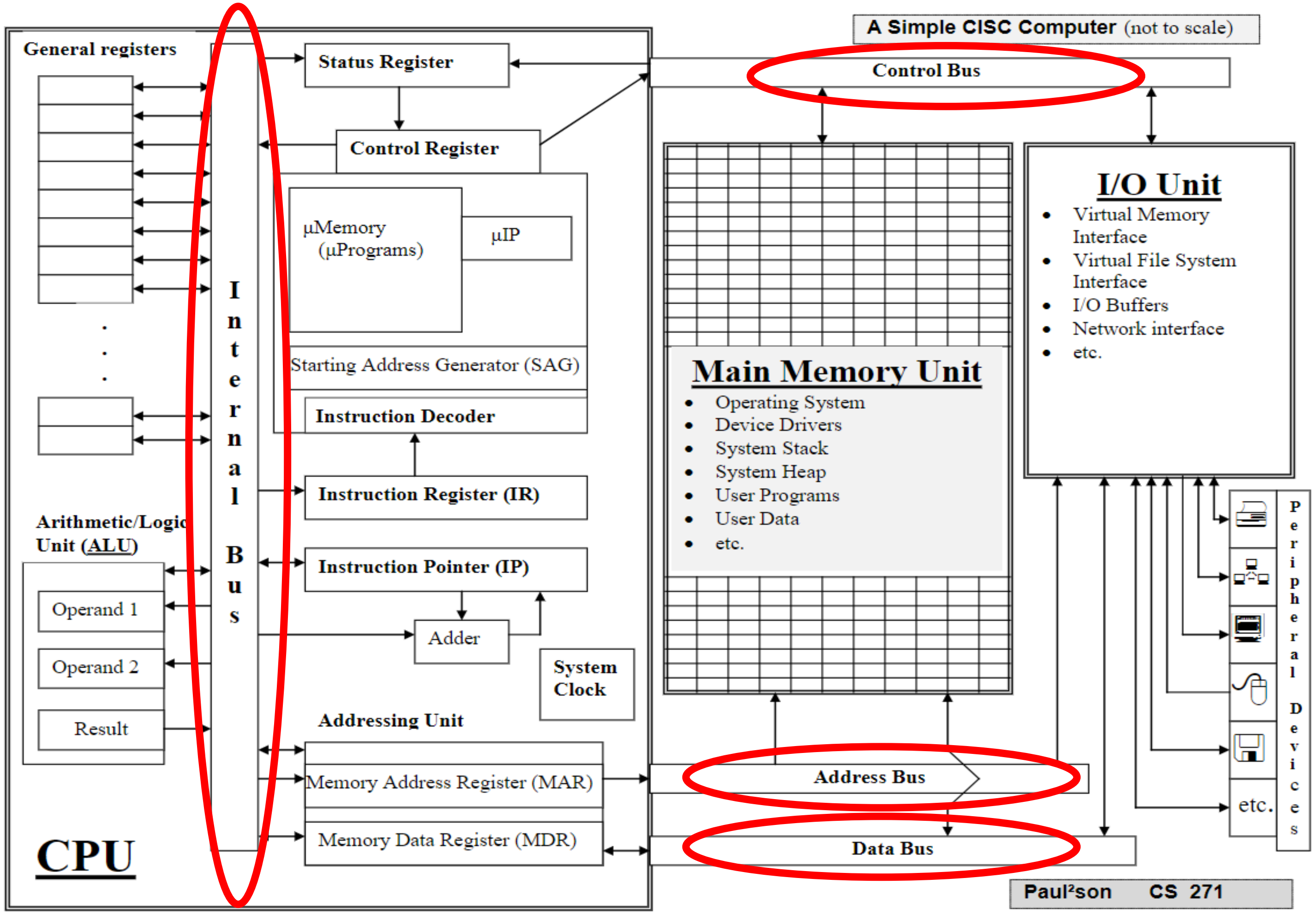
Main parts of the CISC diagram (Complex Instruction Set Computer)

- **CPU**: Central Processing Unit:
  - Execute machine instructions



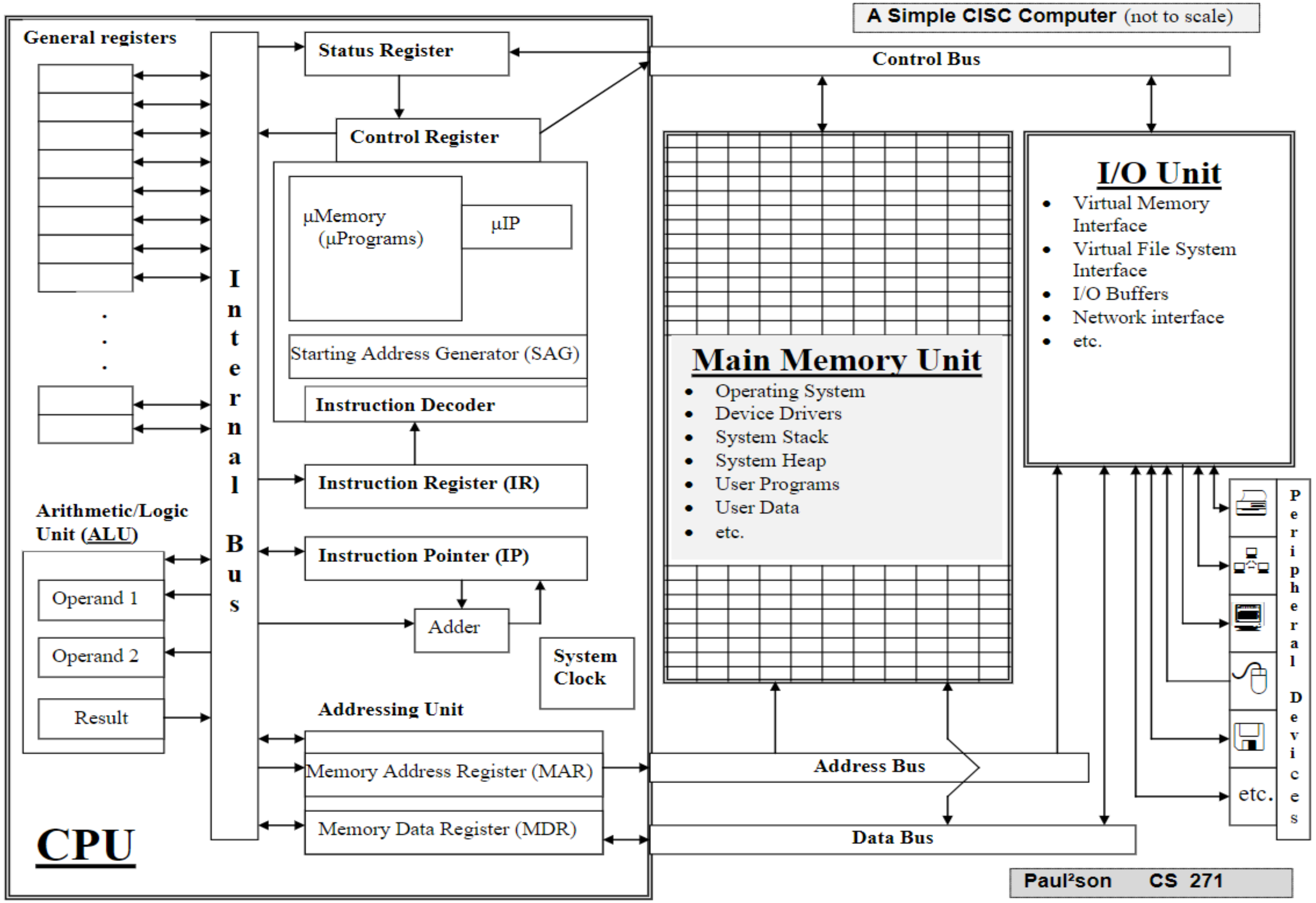
Components / Terms

- **Bus**: parallel "wires" for transferring a set of electrical signals simultaneously
  - Internal: Transfers signals among CPU components
  - Control: Carries signals for memory and I/O operations
  - Address: Links to specific memory locations
  - Data: Carries data CPU ←→ memory



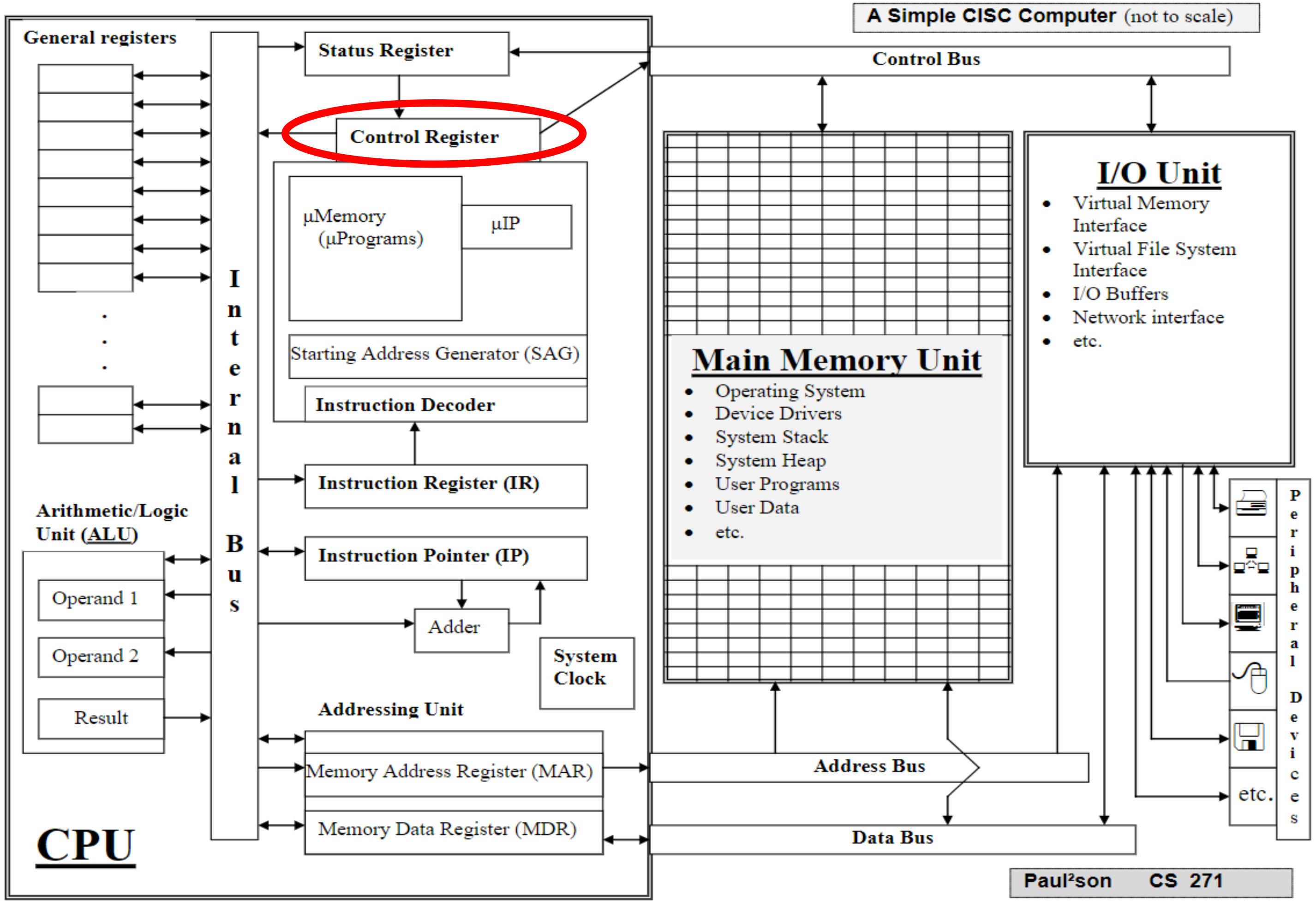
Components / Terms

- **Register**: fast local memory inside the CPU
- **ALU**: Arithmetic Logic Unit
- **Microprogram**: sequence of micro-instructions (implemented in hardware) required to execute a machine instruction
- **Micromemory**: the actual hardware circuits that implement the machine instructions as microprograms



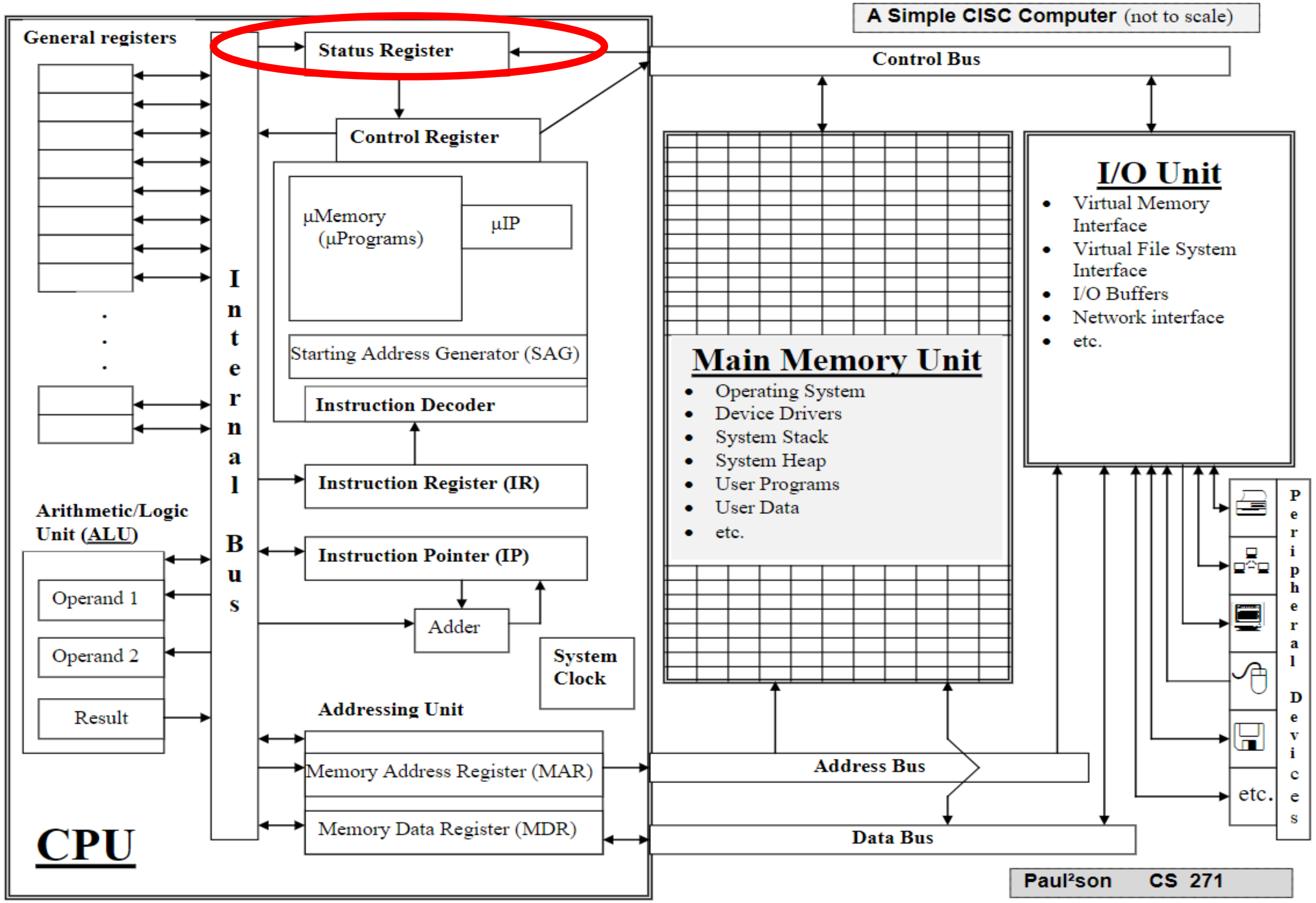
Registers

- **Control**: dictates current state of the machine



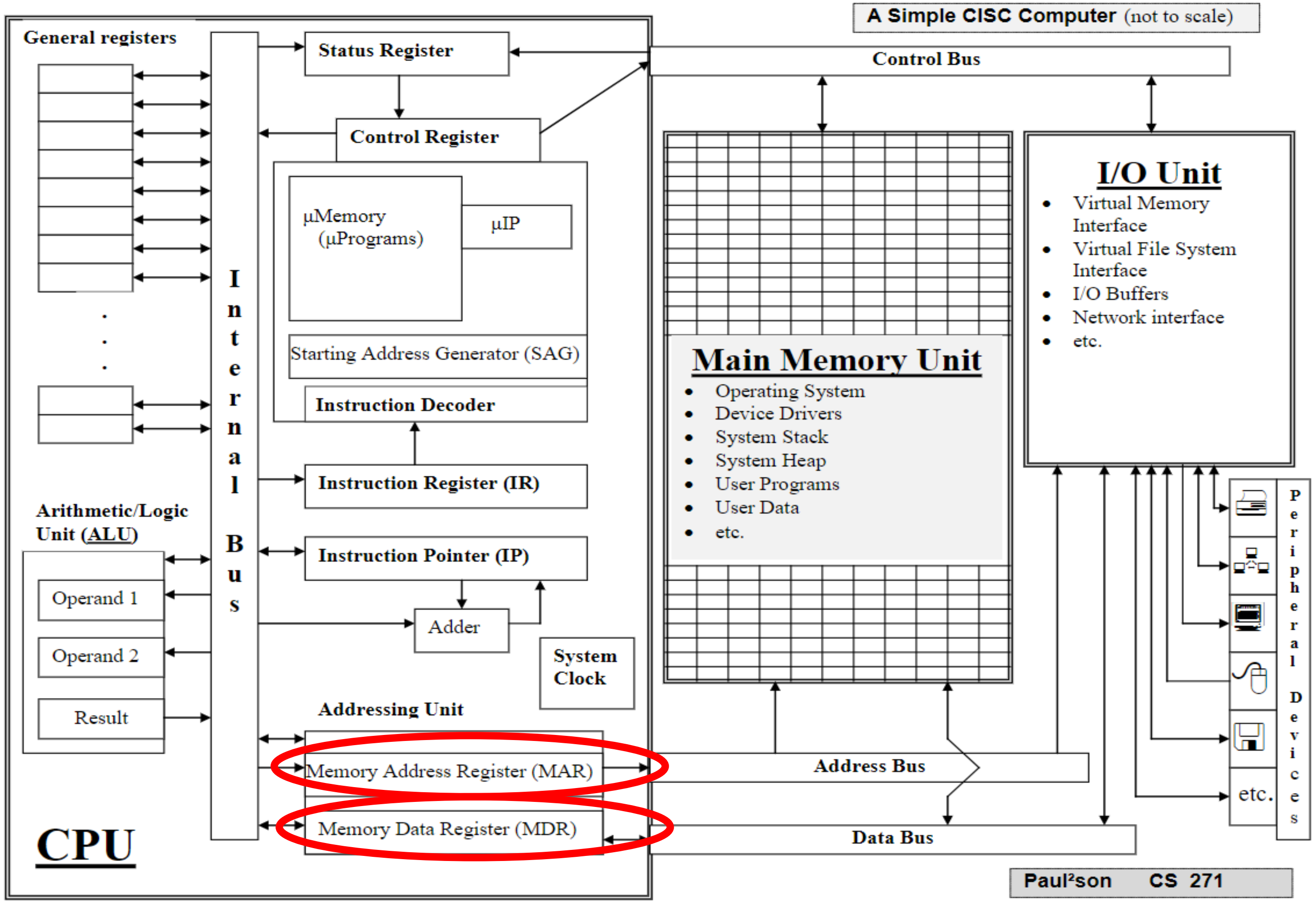
Registers

- **Status**: indicates status of operation (error, overflow…)



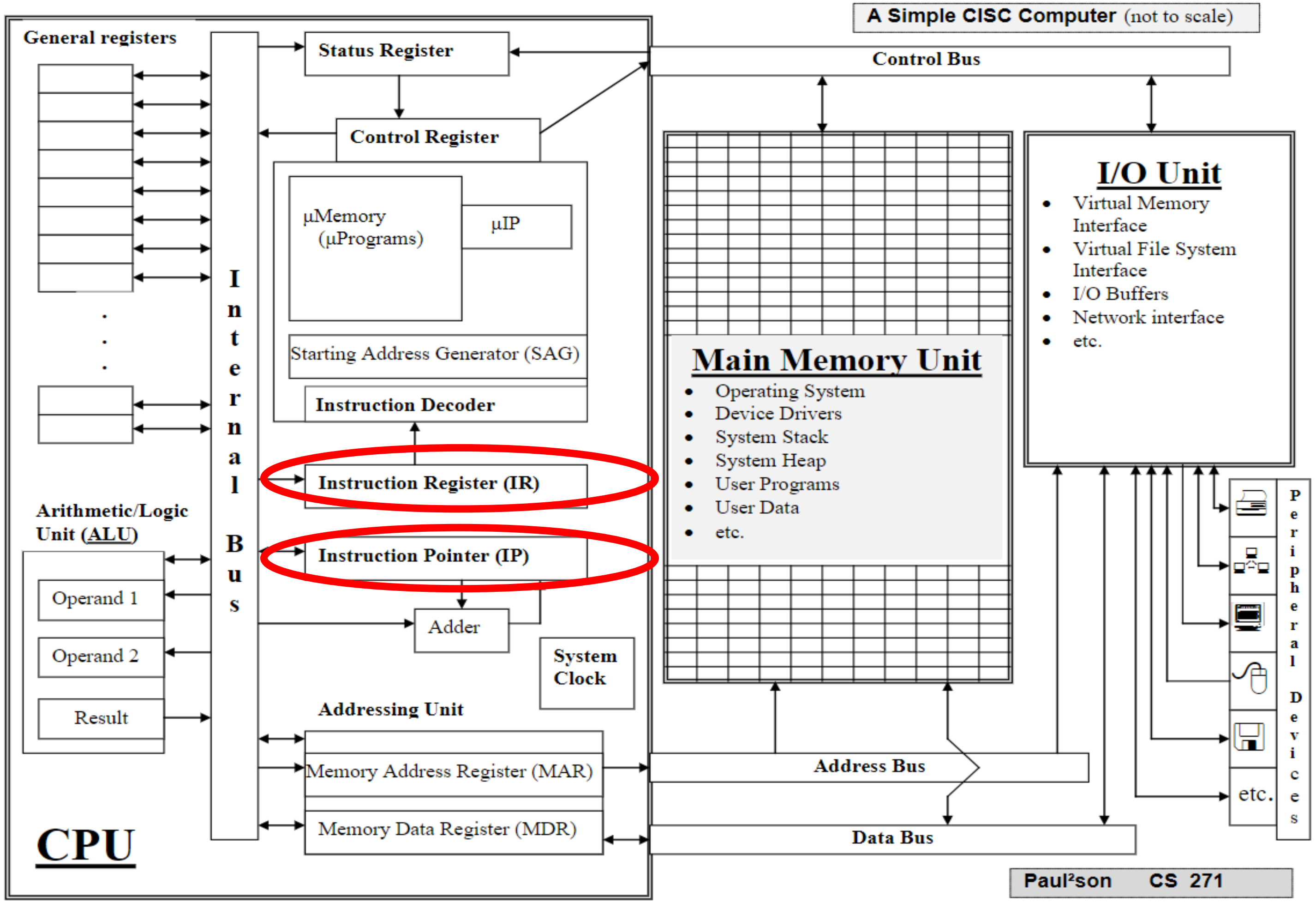
Registers

- **MAR**: Memory Address Register (holds address of memory location currently referenced)
- **MDR**: Memory Data Register: holds data being sent to or retrieved from the memory address in the MAR



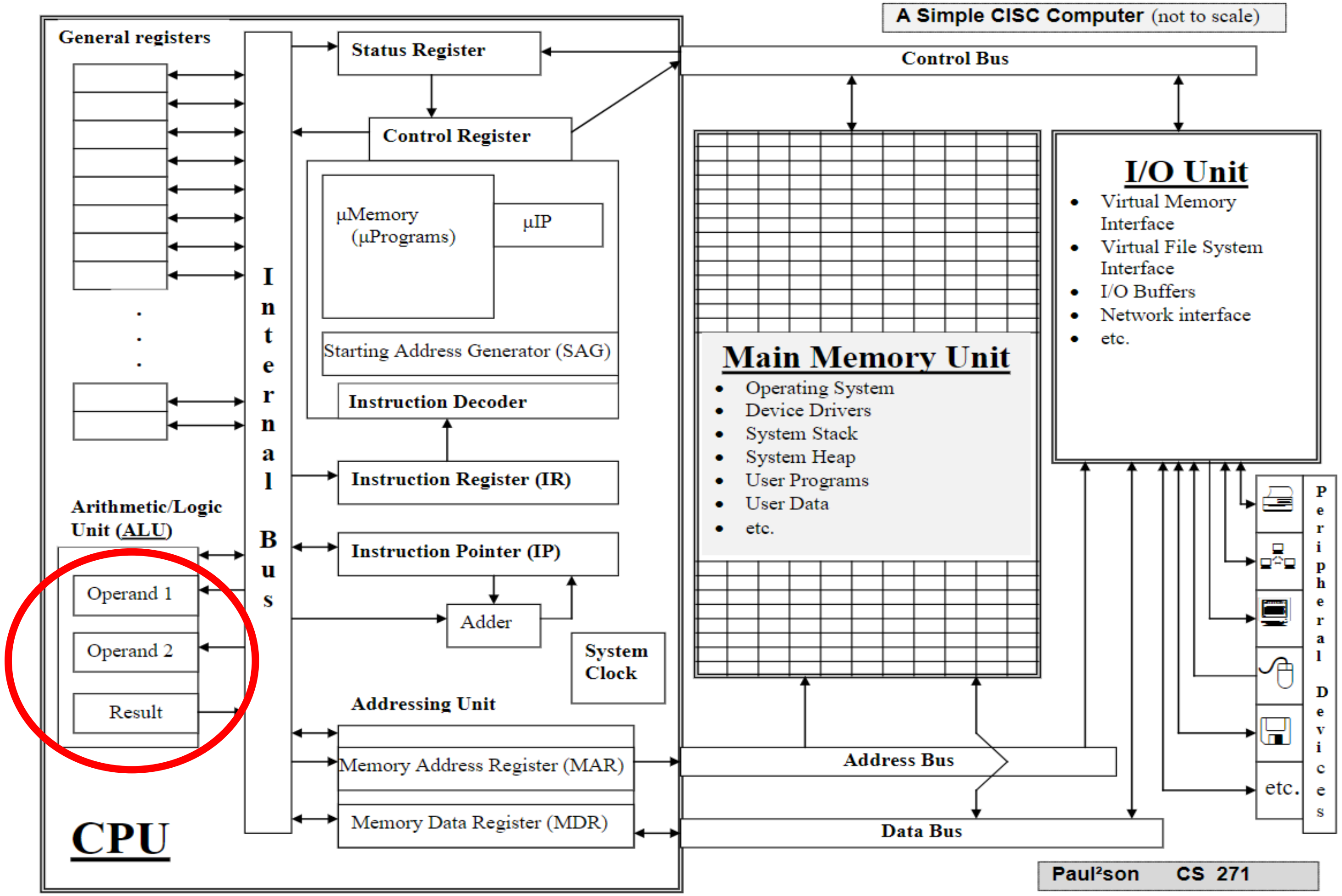
Registers

- **IP**: Instruction Pointer (Holds memory address of next instructions)
- **IR**: Instruction Register (holds current machine instruction)



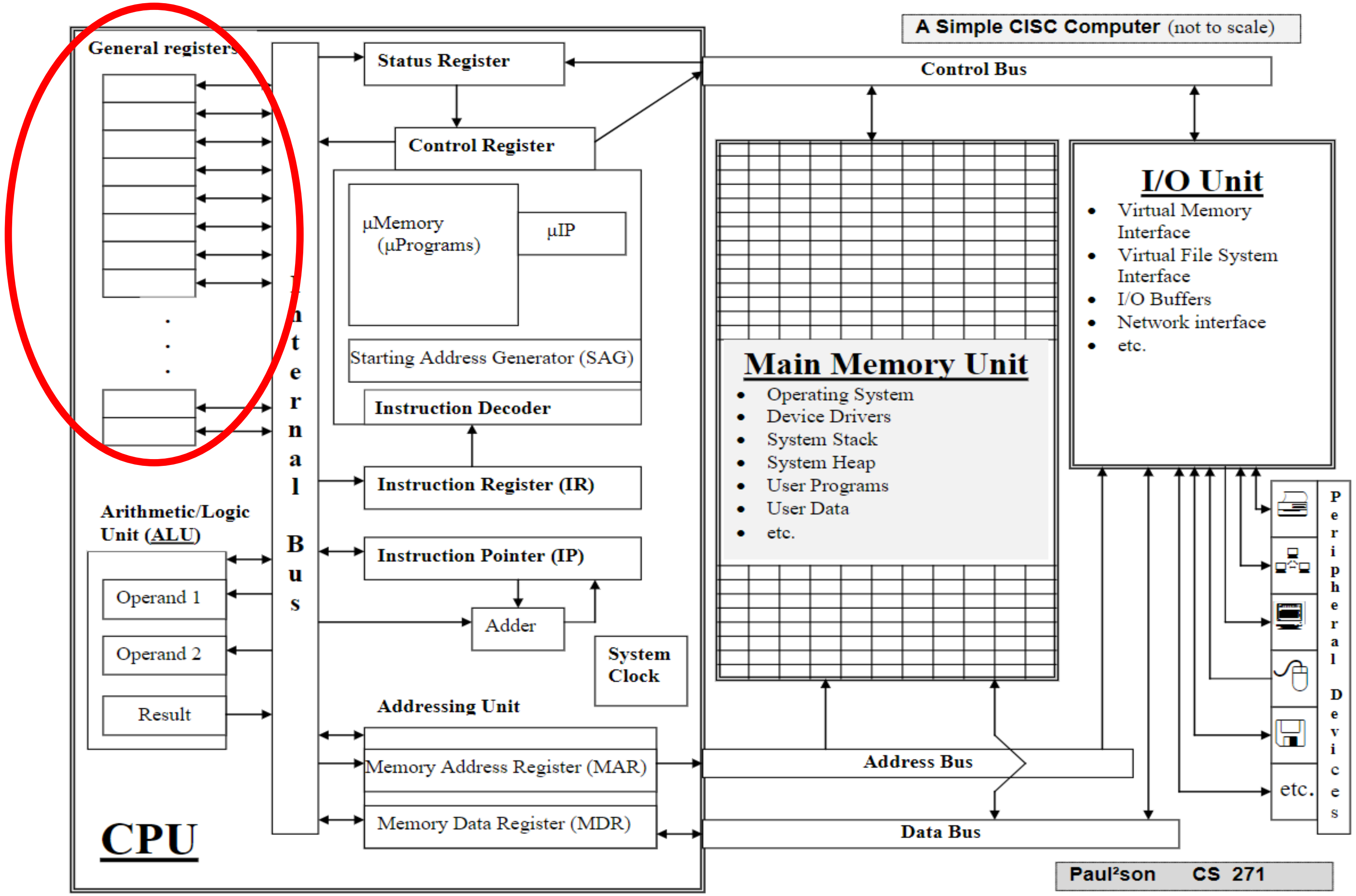
Registers

- **Operand_1, Operand_2, Result**: ALU registers (for calculations and comparisons)



Registers

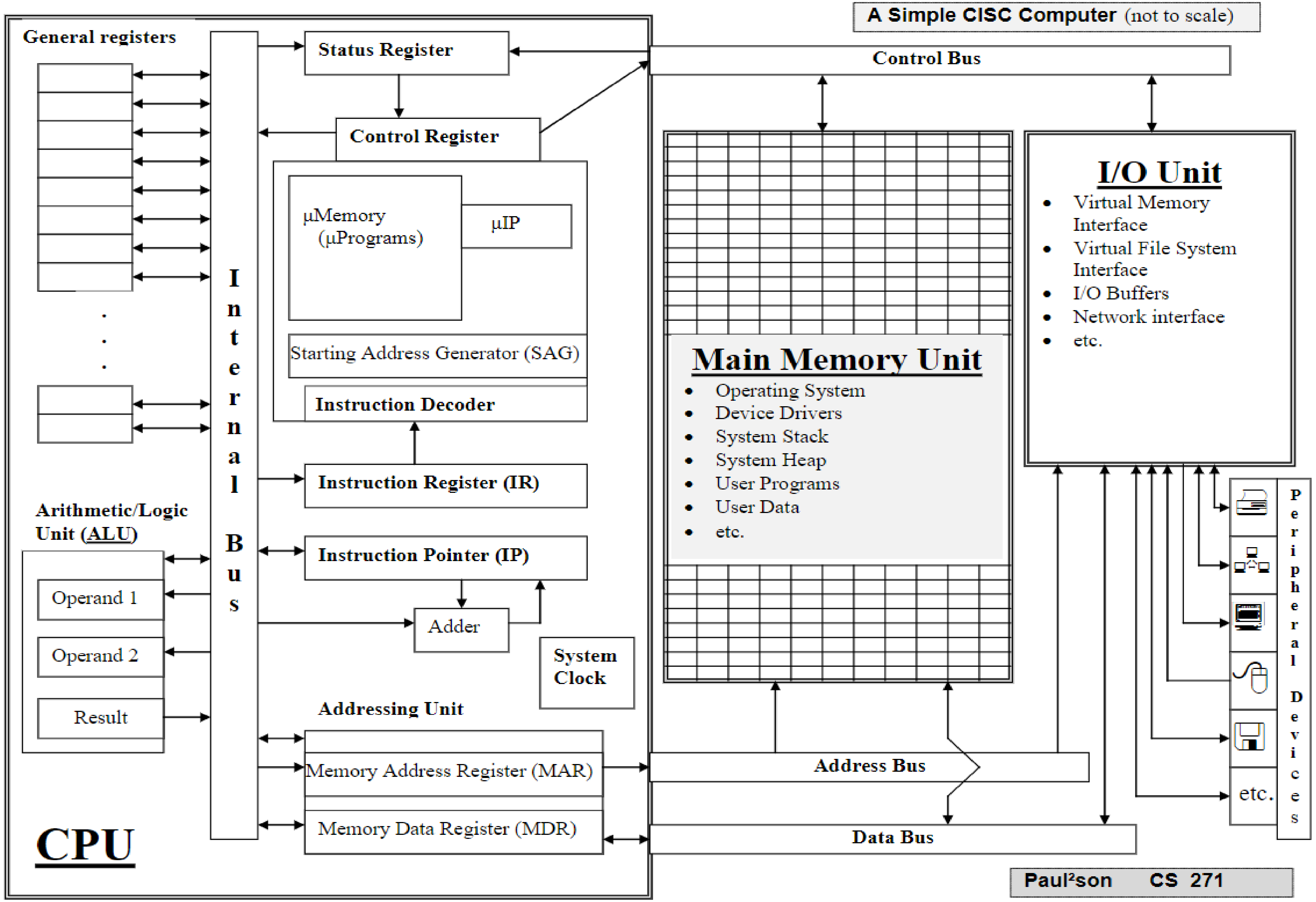
- **General**: fast temporary storage



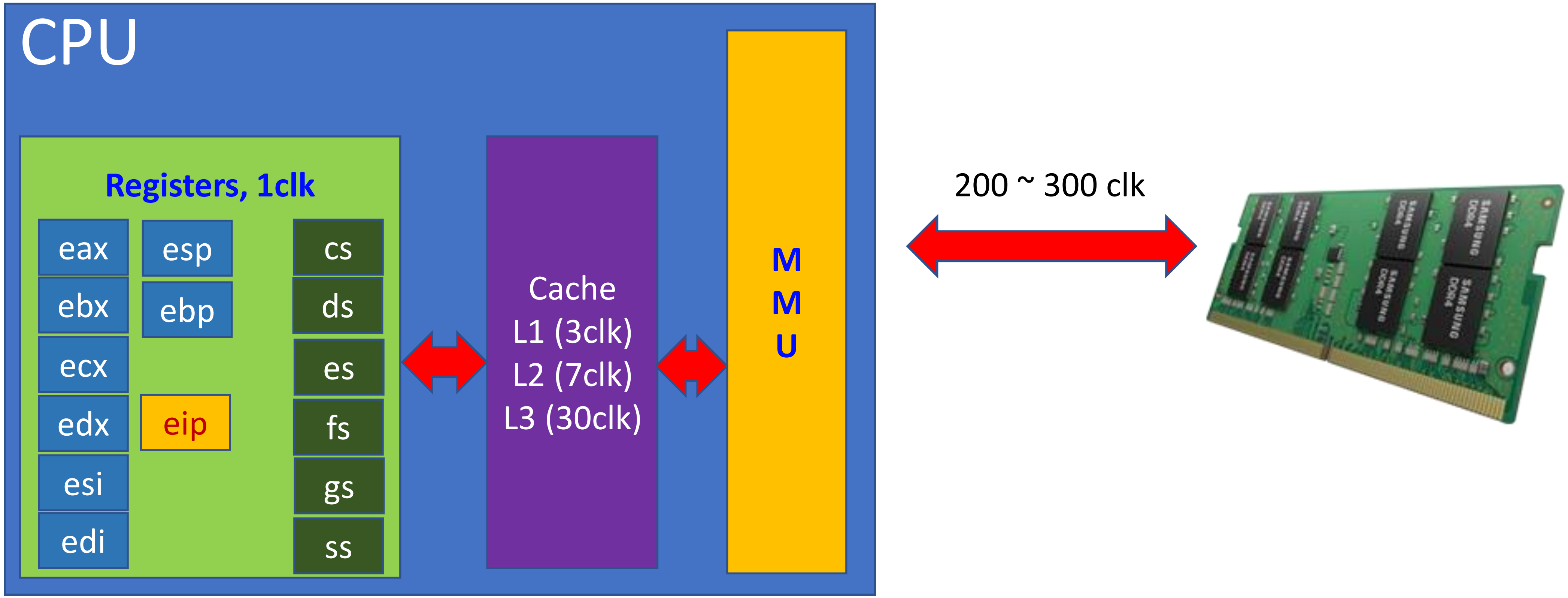
Cache

- **Cache**: an area of comparatively fast temporary storage for information copied from slower storage.
  - Examples:
    - Program instructions are moved from secondary storage to main memory, so they can be accessed more quickly
    - Data is moved from main memory to a CPU register, so it can be accessed instantaneously
- Caching takes places at several levels in a computer system
  - More later on Caching

Cache


A Simple CISC Computer (not to scale)
General registers
Status Register
Control Register
Control Bus
µMemory
(µPrograms)
µIP
Starting Address Generator (SAG)
Instruction Decoder
Instruction Register (IR)
Instruction Pointer (IP)
Adder
System Clock
Internal Bus
Arithmetic/Logic Unit (ALU)
Operand 1
Operand 2
Result
Addressing Unit
Memory Address Register (MAR)
Memory Data Register (MDR)
CPU
Main Memory Unit
• Operating System
• Device Drivers
• System Stack
• System Heap
• User Programs
• User Data
• etc.
I/O Unit
• Virtual Memory Interface
• Virtual File System Interface
• I/O Buffers
• Network interface
• etc.
Peripheral Devices
etc.
Address Bus
Data Bus
Paul²son CS 271

CPU / Registers / Memory



eax General-purpose registers

eip Hidden register. You cannot access it

cs Segment registers, stores CPL/RPL

Real computers …

- … use the “stored program” concept
  - VonNeumann architecture
    - Program is stored in memory, and is executed under the control of the operating system
- … operate using an Instruction Execution Cycle

Instruction Execution Cycle

1. Fetch next instruction (at address in IP) into IR
2. Increment IP to point to next instruction
3. Decode instruction in IR
4. If instruction requires memory access
   A. Determine memory address.
   B. Fetch operand from memory into a CPU register, or send operand from a CPU register to memory.
5. Execute micro-program for instruction
6. Go to step 1 (unless the “halt” instruction has been executed)

Note: default execution is sequential

Example CISC Instruction

```
ADD  R1, mem1
```
;Example assembly language instruction

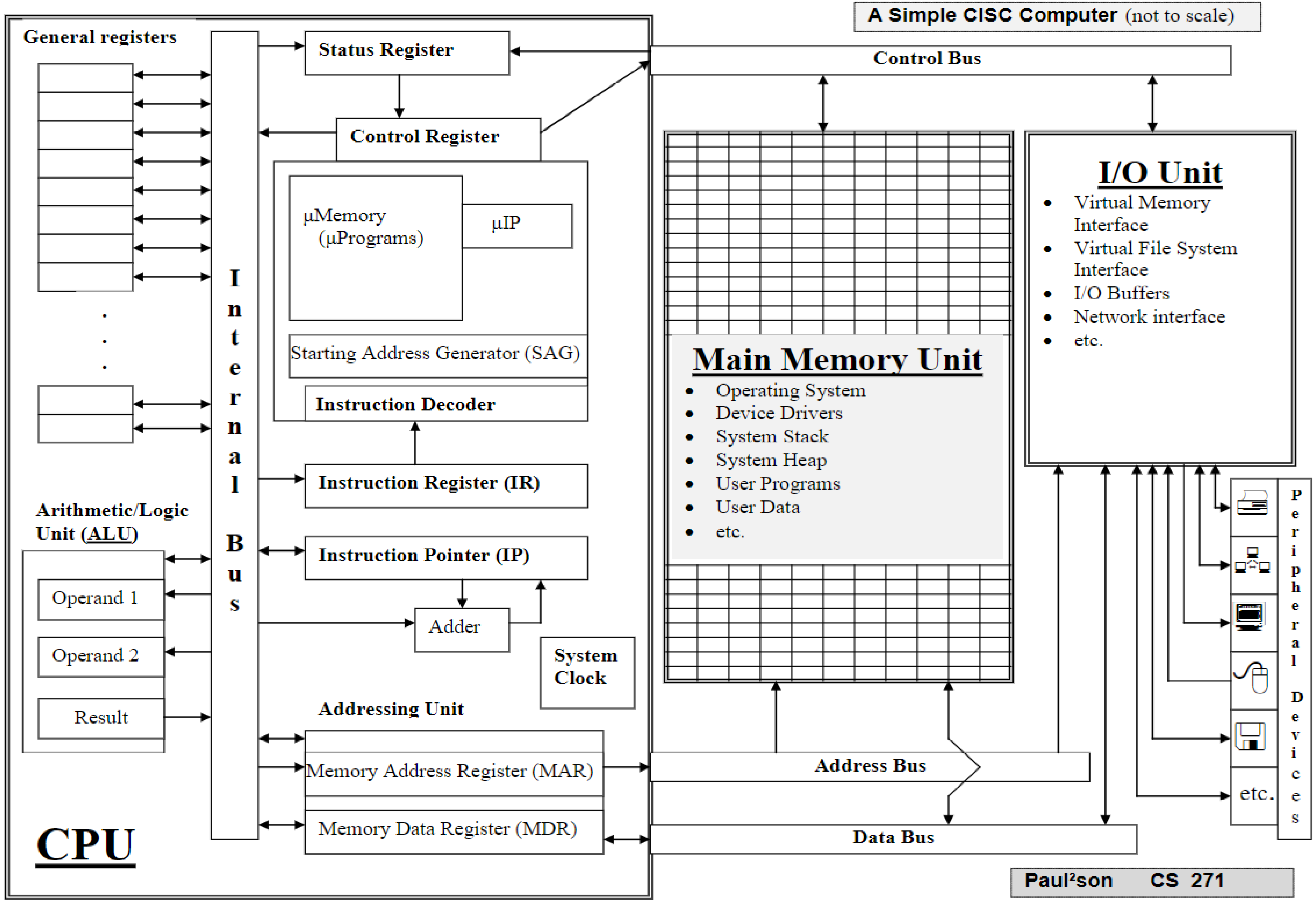
Meaning: Add value in memory location mem1 to value in register R1

Example ADD Microprogram:

(each micro instruction executes in one clock cycle)

1. Copy contents of R1 to ALU Operand_1
2. Move address mem1 to MAR
3. Signal memory fetch (gets contents of memory address currently in MAR into MDR)
4. Copy contents of MDR into ALU Operand_2
5. Signal ALU addition
6. Set Status Register and Copy contents of ALU Result to register R1

Example CISC Instruction



Things get complicated …

- Even in the simplest architectures
  - Bus Arbitration required
  - CPU scheduling required
- As architectures become more complex
  - Multi-processor coordination required
  - Cache management required
- Etc. …

Introduction to Intel IA-32 architecture

Preliminaries: Metrics (measurements)

- Speed (distance/time) is measured in electronic units:
  - K = 10^3, M = 10^6, G = 10^9, etc.
  - e.g. network speed of 8 Mbps means 8,000,000 bits per second

- Size in bits, Bytes is measured in binary units
  - Commonly used: K = 2^{10}, M = 2^{20}, G = 2^{30}, etc.
  - In this course, use: Ki = 2^{10}, Mi = 2^{20}, Gi = 2^{30}
  - e.g., disk size of 200 GiB means
  - 200 * 2^{30} Bytes = 214,748,364,800 Bytes = 1,717,986,918,400 bits
- Bytes and bits (abbreviations)
  - Use lower-case **b** for bits
  - Use upper-case **B** for Bytes
  - Example: 1Mi**b** = 128 Ki**B**

Intel IA-32 Architecture

- CISC
- Two modes of operation:
  - Protected
  - Real-address
- Two processors in one
  - Integer unit
  - Floating-point unit
  - Two processors can work in parallel (co-pressors)
    - Separate instructions sets
    - Separate data registers
      - Different configuration
    - Separate ALUS

Intel IA-32 Architecture

- Specific hardware implementations
  - Registers
  - Memory addressing scheme
- Specific instruction set and microprograms
- Specific assembly languages
  - MASM, NASM, TASM, etc.
- Specific operating systems
  - Windows, Linus, DOS, etc.

Intel IA-32 Architecture

- Memory
  - Up to 4 GiB
  - Byte-addressable
  - Little-endian
- 32-bit machine
  - Registers
  - Buses
  - ALU

Intel IA-32 Architecture

- **Byte** is the smallest unit of data that can be manipulated directly in the IA-32 architecture.
- Operating system and instruction decoder determine how byte codes are interpreted
  - Integer
  - Character
  - Floating-point
  - Instruction
  - Address
  - Status bits

Integer Unit Registers

32-bit **general**-purpose registers	32-bit **multi**-purpose registers
EAX	EBP
EBX	ESP
ECX	ESI
EDX	EDI

32-bit **special**-purpose registers	16-bit **segment** registers	
EFL (status)	CS	ES
EIP (instruction pointer)	SS	FS
In protected mode, the Control Register, Instruction Register, MAR, and MDR are usually hidden	DS	GS

Integer Unit Registers

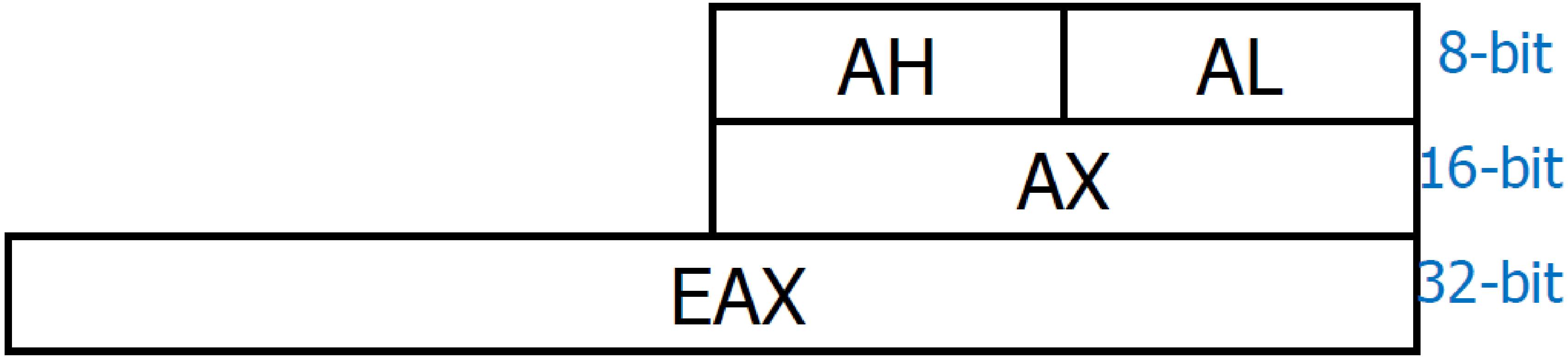
- Most of the 32-bit registers are visible during MASM debugging
  - The 32-bit “general” and “multi” registers may be manipulated directly
  - The 32-bit “special” registers are manipulated by the micro-programs that implement the instructions

Integer Unit Registers

- Some “general-purpose” and “multi-purpose” registers are used for special purposes:
  - EAX and EDX are automatically used by integer multiplication and division instructions
  - ECX is automatically used as a counter for some looping instructions
  - ESP is used for referencing the system stack
  - Etc.

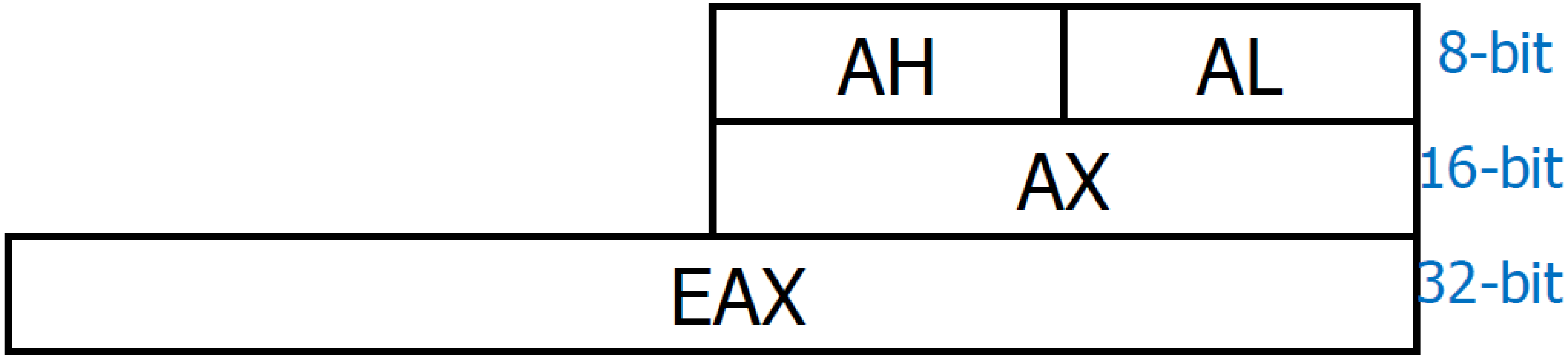
Integer Unit Registers

- Some 32-bit registers have 8-bit and 16-bit “sub-registers”
  - EAX, EBX, ECX, EDX
  - Example: Sub-registers of EAX
    - AX refers to the least-significant 16-bits of EAX
    - AL refers to the least-significant 8-bits of AX
    - AH refers to the most-significant 8-bits of AX



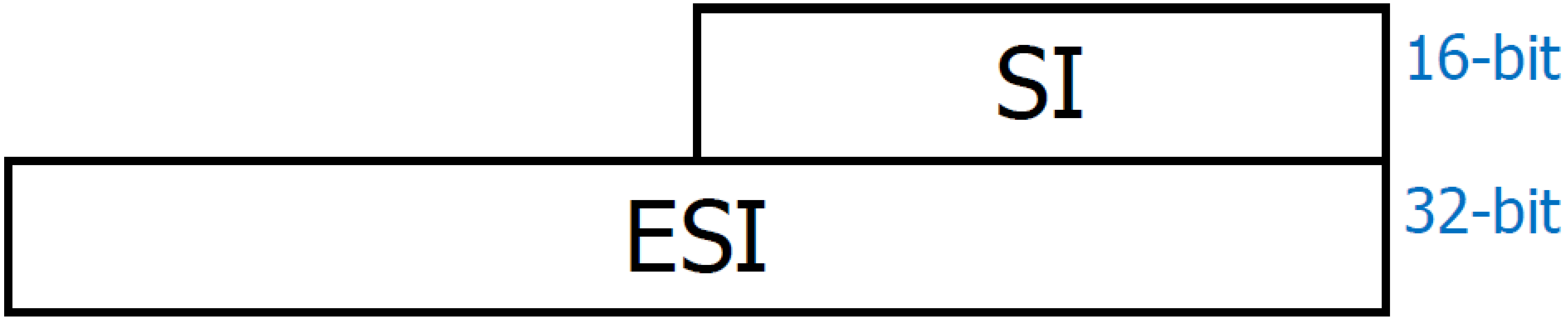
Integer Unit Registers

- <u>Note</u>: if you change a sub-register, the value in the entire register is changed.
- Example:
  - Suppose that EAX contains the electrical representation of **67890**
  - We now give the instruction **`mov AL, 27`**
  - The new value in EAX is 67867



Integer Unit Registers

- Some 32-bit registers have only 16-bit “sub-registers”
  - ESI, EDI, EBP, ESP
  - Example: Sub-registers of ESI
    - SI refer to the least-significant 16-bits of ESI



There's only one set of registers for the integer unit!

- Something like global variables
- Sometimes have to be saved and restored.

- Most register instructions (for now) reference EAX, EBX, ECX, and/or EDX

Introduction to MASM assembly language

MASM Instruction Types

- Move data
- Arithmetic
- Compare two values
- Conditional/unconditional branch
- Call procedure, return
- Loop control
- I/O (input/output)

MASM Directives

- Tell the assembler how to interpret the code
  - Mark beginning of program segments … e.g.

```
.data
.code
```

  - Mark special labels … e.g.

```
main        proc
varName     DWORD
```

  - Etc.

MASM Program Template

```
TITLE Program Template (template.asm)
; Author:
; Course/project ID                                    Date:
; Description:

INCLUDE Irvine32.inc
        <insert constant definitions here>
.data
        <insert variable definitions here>

.code
main PROC
        <insert executable instructions here>
        exit                            ; exit to operating system
main ENDP
        <insert additional procedures here>
END main
```

MASM Programming

- **`TITLE`** directive
  - You can put anything you want
  - … but the grader wants to see a meaningful title and the name of the source code file
- ; identification block
  - Technically optional (as are all comments)
  - … but the grader wants to see information
- **`INCLUDE`** directive
  - Copies a file of definitions and procedures into the source code
  - Use **`Irvine32.inc`** for now

MASM Programming

- `Global constants` may be defined
- `.data` directive
  - Marks beginning of data segment
  - Variable declarations go here
- `.code` directive
  - Marks end of data segment and beginning of code segment
  - `main` procedure defined here (required)
    - Other procedures defined here (optional)
    - `main` must have an `exit` instruction
    - All procedures require `PROC` and `ENDP` directives
- `END` directive
  - Tells operating system where to begin execution

MASM syntax and style

- MASM is **not** case-sensitive!!
  - Constants usually ALL CAPS
- Segments start with `.`
  - `main` should be the first procedure in the `.code` segment
  - Beginning of next segment (or `END main`) is end of segment
- Comments start with `;`
  - Can start anywhere in a line
  - Remainder of line is ignored by the assembler
  - End of line is end of comment
- Use indentation and sufficient white space to make sections easy to find and identify

MASM identifier syntax

- Identifiers: Names for variables, constants, procedures, and labels
- 1 to 247 characters (no spaces)
  - Use concise, meaningful names
- Not case sensitive!
- Start with letter, _, @, or $
  - For now, start with letter only
- Remaining characters are letters, digits, or _
- Cannot be a reserved word
  - E.g.: proc, main, eax, … etc.

Memory Locations

- May be named
  - Name can refer to a variable name or a program label
- Interpretation of contents **<u>depends on program instructions</u>**
  - Numeric data
    - Integer, floating point
  - Non-numeric data
    - Character, string
  - Instruction
  - Address
  - etc.

MASM data types syntax

Type	Used for:
BYTE	Character, string, 1-byte integer
WORD	2-byte integer, address
DWORD	4-byte unsigned integer, address
FWORD	6-byte integer
QWORD	8-byte integer
TBYTE	10-byte integer
REAL4	4-byte floating-point
REAL8	8-byte floating-point
REAL10	10-byte floating-point

MASM Data definition syntax

- In the `.data` segment
- General form is

```
label          data_type          initializer
;comment
```

  - `label` is the "variable name"
  - `data_type` is one of (see previous slides)
  - At least one `initializer` is required
    - May be ? (value to be assigned later)
- Examples:

```
size      DWORD 100           ;class size
celsius   WORD  -10           ;current Celsius temp
response  BYTE  'Y'           ;positive answer
myName    BYTE  "Wile E. Coyote",0
gpa       REAL4 ?             ;my GPA
```

Data in Memory

- "variables" are laid out in memory <u>in the order declared</u>
- Example:

```
.data
size       DWORD  100              ;class size
celsius    WORD   -10              ;current Celsius
response   BYTE   'Y'              ;positive answer
myName     BYTE   "Wile E. Coyote",0
gpa        REAL4  ?                ;my GPA
```

  - Suppose that the data segment starts at memory address 1000

size	is address 1000	(DWORD uses 4 bytes)
celsius	is address 1004	(WORD uses 2 bytes)
Response	is address 1006	(BYTE uses 1 byte)
myName	is address 1007	(Each character uses 1 byte)
		(Blank spaces and the terminating 0 are characters too!)
gpa	is address 1022	

Data in Memory

`size`	is address 1000	(DWORD uses 4 bytes)
`celsius`	is address 1004	(WORD uses 2 bytes)
`Response`	is address 1006	(BYTE uses 1 byte)
`myName`	is address 1007	(Each character uses 1 byte)
		(Blank spaces and the terminating 0 are characters too!)
`gpa`	is address 1022	

- Note:
- Each **name** is a **constant**
  - i.e. the system substitutes the memory address for each occurrence of a name
- The **contents** of a memory location may be **variable**.

Literals

- Actual values, named constants
  - Integer
  - Floating point
  - Character
  - String (only in `.data` segment or named constant)
- Used for:
  - Initializing variables (in the `.data` segment)
  - Defining constants
  - Assigning contents of registers
  - Assigning contents of memory (in the `.code` segment)

MASM Literals syntax

- Integer
  - Optional radix: b, q/o, d, h
    - Digits must be consistent with radix (e.g., 1011b, 235q, 2012d, 30h)
    - Hex values that start with a letter must have a leading 0 (e.g., 0A3h)
      - Or use the 0x prefix instead of the radix (e.g., 0xA3)
  - Default is <u>decimal</u>
- Floating-point (decimal real)
  - Optional sign
  - Standard notation (e.g., -3.5 +5. 7.2345)
  - Exponent notation (e.g., -3.5E2 6.15E-3)
  - <u>Must have a decimal point</u>

MASM Literals syntax

- Character
  - Single character in quotes
    - `‘a’` `“*”` `‘3’`
    - Single quotes recommended
- String
  - 2 or more characters in quotes
    - `“always”, 0`
    - `‘123 * 456’, 0`
    - Double quotes recommended
    - Embedded quotes must be different
      - `“It’s”, 0` `‘Title: “MASM”’, 0`
  - String must be null-terminated
    - Always end with zero-byte

MASM Instruction syntax

- Each instruction line has 4 fields:
  - Label
  - Opcode
  - Operands
  - Comments
- Depending on the opcode, one or more operands may be required
  - Otherwise, any field may be empty
  - If empty opcode field, operand field must be empty

MASM Instruction syntax

- Opcode (specifies what to do)
  - Mnemonic (e.g., ADD, MOV, CALL, etc.)
- Zero, one, or two Operands (specify the opcode's target)
  - Different number of operands for different opcodes

```
opcode
opcode    destination
opcode    destination, source
```

MASM Addressing modes

Specific “addressing modes” are permitted for the operands associated with each opcode.

- Basic (used in first programming assignment)
  - Immediate — Constant, literal, absolute address
  - Register — Contents of register
  - Direct — Contents of referenced memory address
  - Offset — Memory address; may be calculated
- Advanced (used in later assignments)
  - Register indirect — Access memory through address in a register
  - Indexed — “array” element, using offset in register
  - Base-indexed — start address in one register; offset in another, add and access memory
  - Stack pointer — Memory area specified and maintained as a stack; stack in ESP register

See the MASM list of instructions

Writing a MASM program

- Demo

Example Problem Definition

Write a MASM program to perform the following tasks:

1. Introduce yourself to the user.
2. Get the user’s name and age.
3. Greet the user, and report the user’s age in dog years.
4. Say good-bye to the user.

Requirements:

1. The user’s name and age must be entered by the user, and must be stored and accessed as data segment variables.
2. The “dog-years factor” (7) must be defined as a constant.

Program Design

- Decide what the program should do
- Define algorithm(s)
- Decide what the output should show
- Determine what variables/constants are required

Implementing a MASM program

- Open project
- Start with template, “save as” <.asm file in the program directory>
  - This is the source code file
- Fill in identification block information
- Create comment outline for algorithms
- Define constants
  - Test/fix (syntax check, nothing happens)
- Declare variables (.data section)
  - Test/fix (syntax check, nothing happens)
- Enter the output code
  - Test/fix (no calculations, usually everything show 0)
- Enter the input code
  - Test/fix (no calculations, echo input)
- Enter the calculation code
  - Test/fix (logic check, verify)

*First try Debug, Start Without Debugging (more later on using the debug system)

Writing a MASM program

- Rules & Regulations
- Syntax and semantics

MASM Instructions

- For now, know how to use
  - mov, add, sub, mul, div, call
- Some instructions use implied operands
- See textbook (Appendix) or on-line instructions

Easy Instructions

- For 2-operand instructions, the 1st operand is the destination, and the 2nd operand is the source
- 2-operand instructions require at least one of the operands to be a register (or op2 must be literal).
  - Note: op1 cannot be a literal

```
mov     op1, op2     ;op2 is copied to op1
add     op1, op2     ;op2 is added to op1
sub     op1, op2     ;op2 is subtracted from op1
inc     op1          ; add 1 to op1
dec     op1          ; subtract 1 from op1
```

Instructions with implied operands

- mul implied operand must be in EAX
- mul op2 ; result is in EDX:EAX
- Example:

```
mov     eax, 10
mov     ebx, 12
mul     ebx              ; result is in eax (120)
                         ; with possible overflow in edx
                         ; edx is changed!
```

Instructions with implied operands

- div implied operand is in EDX:EAX
- So extend EAX into EDX before division
- div op2 ; quotient is in EAX

  ; remainder is in EDX

- Example:

```
mov     eax, 100
cdq                     ; extend the sign into edx
mov     ebx, 9
div     ebx             ; quotient is in eax (11)
                        ; remainder is in edx (1)
```

Operand notation (See Instruction list)

Operand	Description
r8	8-bit general-purpose register: AL, AH, BL, BH, CL, CH, DL, DH
r16	16-bit general-purpose or multi-purpose register: AX, BX, CX, DX, SI, DI, BP, SP
r32	32-bit general-purpose or multi-purpose register: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
reg	any general-purpose or multi-purpose register
accum	AL, AX, or EAX (depending on operand size)
mem	8-bit, 16-bit, or 32-bit memory location (depending on operand size)
segreg	16-bit segment register: SS, CS, DS, ES, FS, GS
r/m8	8-bit register or memory location
r/m16	16-bit register or memory location
r/m32	32-bit register or memory location
imm8	8-bit literal value
imm16	16-bit literal value
imm32	32-bit literal value
imm	8-bit, 16-bit, or 32-bit literal value (depending on operand size)

Examples

Syntax	Examples
MOV *mem,accum*	`mov total,eax` `mov response,al`
MOV *accum,mem*	`mov al,char` `mov eax,size`

Syntax	Examples
MOV *mem,imm*	`mov color,7` `mov response,'y'`

Syntax	Examples
MOV *reg,imm*	`mov ecx,256` `mov edx,OFFSET myString`

Notes:

accum means "eax or some valid part of eax"

imm means "a literal or constant"

Examples

Syntax	Examples

MOV *reg,reg*	`mov dh,bh`
	`mov edx,ecx`
	`mov ebp,esp`

MOV *mem,reg*	`mov count,ecx` `mov num1,bx`
MOV *reg,mem*	`mov ebx,pointer` `mov al,response`

Notes:

mem8 means "BYTE"

mem16 means "WORD"

mem32 means "DWORD"

sreg means CS, DS, ES, FS, GS, or SS

Syntax	Examples
MOV *sreg,reg16*	`mov ds,ax`
MOV *sreg,mem16*	`mov es,pos1`
MOV *reg16,sreg*	`mov ax,ds`
MOV *mem16,sreg*	`mov stack_save,ss`

Invalid MOV statements

```
.data
bVal   BYTE    100
bVal2  BYTE    ?
wVal   WORD    2
dVal   DWORD   5

.code
   mov ds,45         immediate move to DS not permitted
   mov esi,wVal      size mismatch
   mov eip,dVal      EIP cannot be the destination
   mov 25,bVal       immediate value cannot be destination
   mov bVal2,bVal    memory-to-memory move not permitted
```

Libraries

- We will use Irvine’s library (for now) to handle the really awful stuff
  - Input/output
  - Screen control
  - Timing
  - etc.
- Check IrvineLibHelp, or find the descriptions in your textbook.

Library Procedures – Overview 1

- Clrscr – clear the screen
  - Preconditions: none
  - Postconditions: screen cleared, and cursor is at upper left corner

- Crlf – New line
  - Preconditions: none
  - Postconditions: cursor is at beginning of next new line

Library Procedures – Overview 2

- ReadInt – Reads an integer from keyboard, terminated by the Enter key
  - Preconditions: none
  - Postconditions: value entered is in EAX

- ReadString – Reads a string from keyboard, terminated by the Enter key
  - Preconditions: OFFSET of memory destination in EDX
    Size of memory destination in ECX
  - Postconditions: String entered is in memory
    Length of string entered is in EAX

Library Procedures – Overview 3

- WriteInt, WriteDec – Writes an integer to the screen
  - Preconditions: value in EAX
  - Postconditions: value displayed
  - WriteInt displays +/-

- WriteString – Writes a null-terminated string to the screen
  - Preconditions: OFFSET of memory location in EDX
  - Postconditions: String displayed

Calling a Library Procedure

- The INCLUDE directive copies the procedure prototypes (declarations) into the program source code.

- Call a library procedure using the CALL instruction.

In-line Comments

- Start with ;
- May be on separate line or at the end of a line
- Use comments to clarify lines or sections
- Preferred …

```
; Calculate the number of students on-line today.
        mov     eax,size
        sub     eax,absent
        mov     present,eax
```

- OK …

```
        mov     eax,size        ;start with class size
        sub     eax,absent      ;subtract absentees
        mov     present,eax     ;number present
```

- Terrible …

```
        mov     eax,size        ;move size into eax
        sub     eax,absent      ;subtract absent from eax
        mov     present,eax     ;move eax to present
```