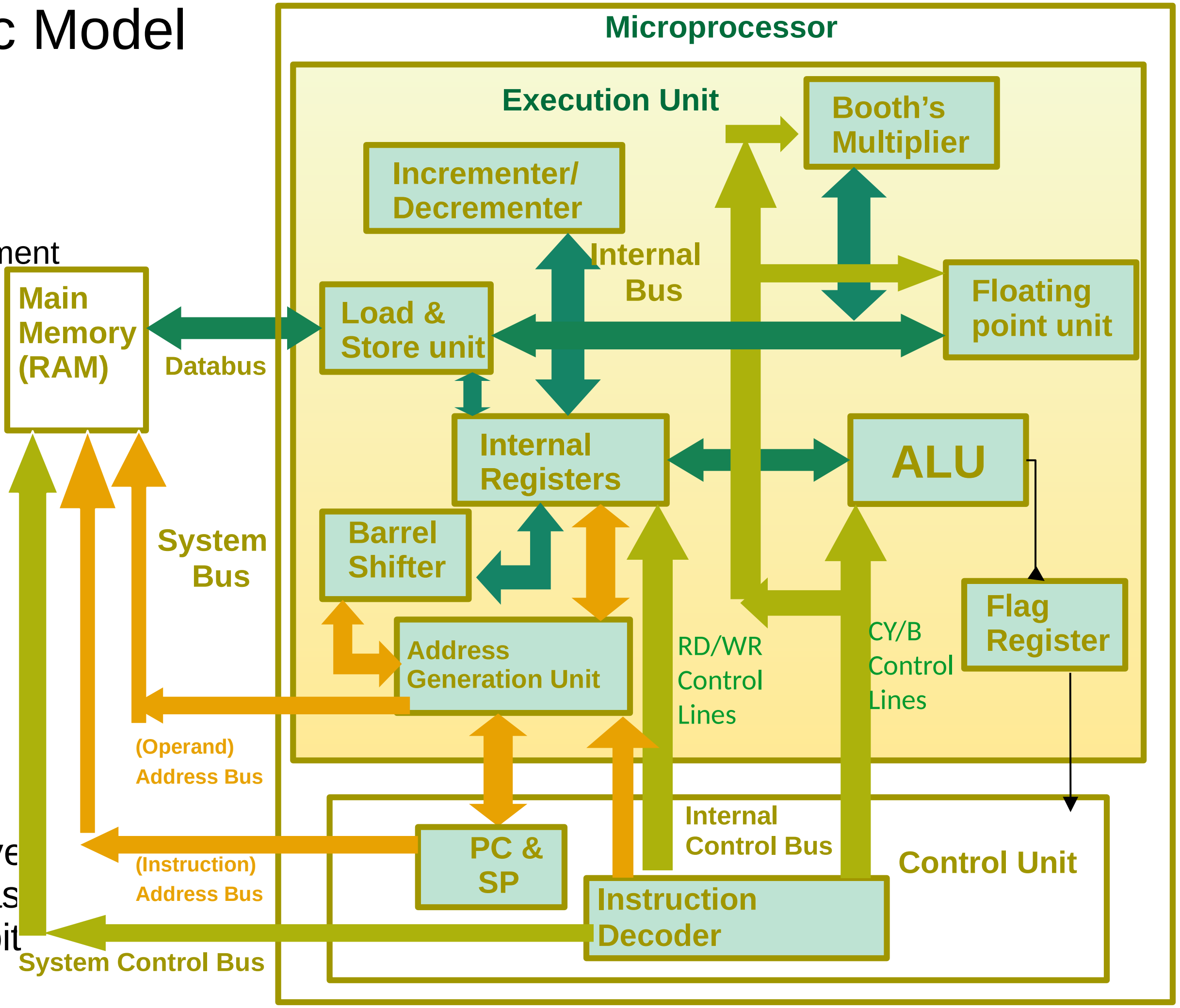
EE2016 Microprocessor Theory & Lab
Aug – Nov 2023

Week5: Execution Unit – Deeper Look

Dr. R. Manivasakan, OWSM, EED, IIT, Madras

Engines in Execution Unit: Generic Model

- ALU
  - ➢ Arithmetic Operations
    - ADD, ADD with carry, SUB, SUB with borrow, 2’s complement, 1’s complement, increment / decrement
  - ➢ Logical Operations
    - AND, OR, Ex-OR
- Shifting (Barrel shifter)
  - Arithmetic shift
  - Logical shift
  - Rotate
  - Rotate through carry
- Address Generation Unit
  - ➢ Calculates the addresses in main memory as requested by control unit
  - ➢ Those address-generation calculations involve different integer arithmetic operations, such as addition, subtraction, modulo operations, or bit shifts.



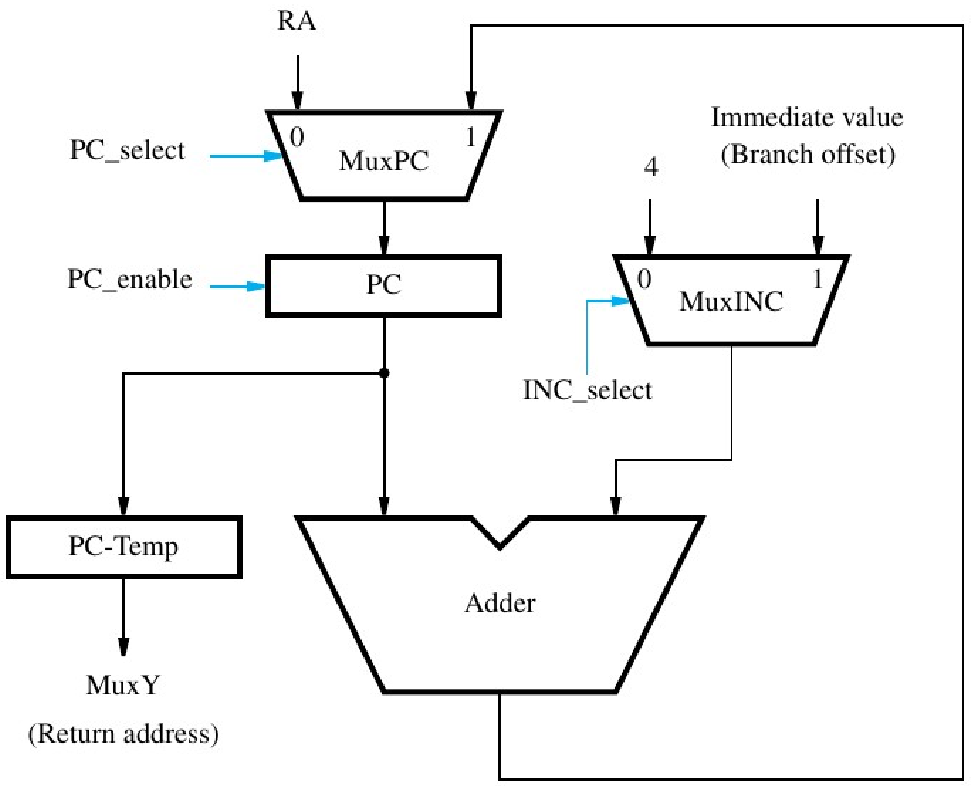
Instruction Address Generator

Example Program

Ref Hamachar 6e Fig 5:10 in page 165

- Instruction Address Generator (IAG)
  - Engine which determines the address of the instruction to be executed next (esp. in ‘jump’ instance of program flow)
- Jump in program flow through branch offset value or a jump of fixed number of instructions
  - One adder input is connected to the PC. The second input is connected to a multiplexer, MuxINC, which selects either the constant 4 (unconditional fixed jump) or the branch offset to be added to the PC. The branch offset is given in the immediate field of the IR and is sign-extended to 32 bits by the Immediate block.
- The output of the adder is routed to the PC via a second multiplexer, MuxPC, which selects between the adder and the output of register RA.
  - Register RA input to MuxPC corresponds to, executing subroutine linkage instructions – meaning the PC value has to switch from the address of calling instruction in the mother program to the address of first instruction in the called, viz., subroutine program.
    - RA holds the address of first line in the in the called subroutine or interrupt program
  - Before the above switching, the PC value is stored in PC-Temp. Actually, the PC value stored in PC-Temp, is the subroutine or interrupt return address & this register is actually the stack pointer in modern muPs.
- Control signals
  - The INC_select signal selects the value to be added to the PC, either the

  constant 4 or the branch offset specified in the instruction.
  - The PC_select signal selects either the updated address or the contents of register RA to be loaded into the PC when the PC_enable control signal is activated.



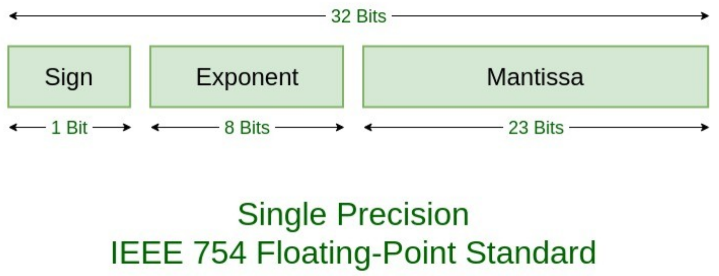
Shifter Applications

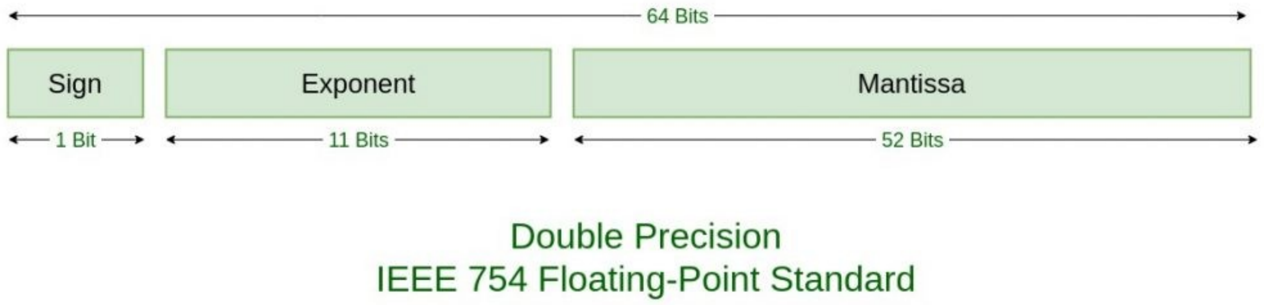
- Floating Point Unit
  - a part of a computer system specially designed to carry out operations on floating-point numbers.
  - Typical operations are addition, subtraction, multiplication, division, and square root.
- Applications of Shifting
  - Fast digital convolutions using bit-shifts
  - Multiplication by 2^n
  - Division by 2^n
  - circular shift is a special kind of cyclic permutation
  - Cyclic codes in generating cyclic codes in secure communications
  - Applications in cyclic language and formal language theory
  - Context free language, then shift is again context free.

- Applications of Shifting (cont'd)
  - For a floating-point add or subtract operation, the significands of the two numbers must be aligned, which requires shifting the smaller number to the right, increasing its exponent, until it matches the exponent of the larger number.
  - This is done by subtracting the exponents and using the barrel shifter to shift the smaller number to the right by the difference, in one cycle. If a simple shifter were used, shifting by n bit positions would require n clock cycles.
- Bit Operations
  - Bit by bit AND operations
    - To check if the given word is even
      - Divisible by 2?
        - Right shift?
  - A
- S

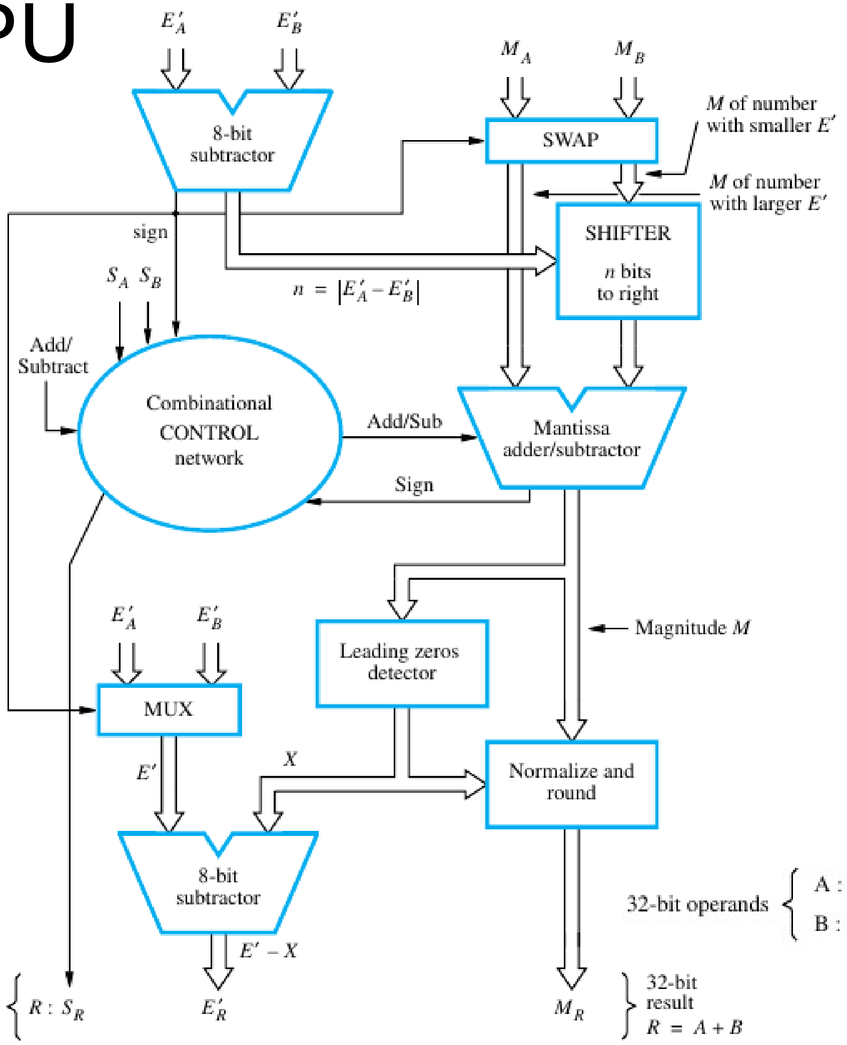
Chandra, Shekhar S. "Fast Digital Convolutions using Bit-Shifts." arXiv preprint arXiv:1005.1497 (2010)

Representation of Floating Point Numbers

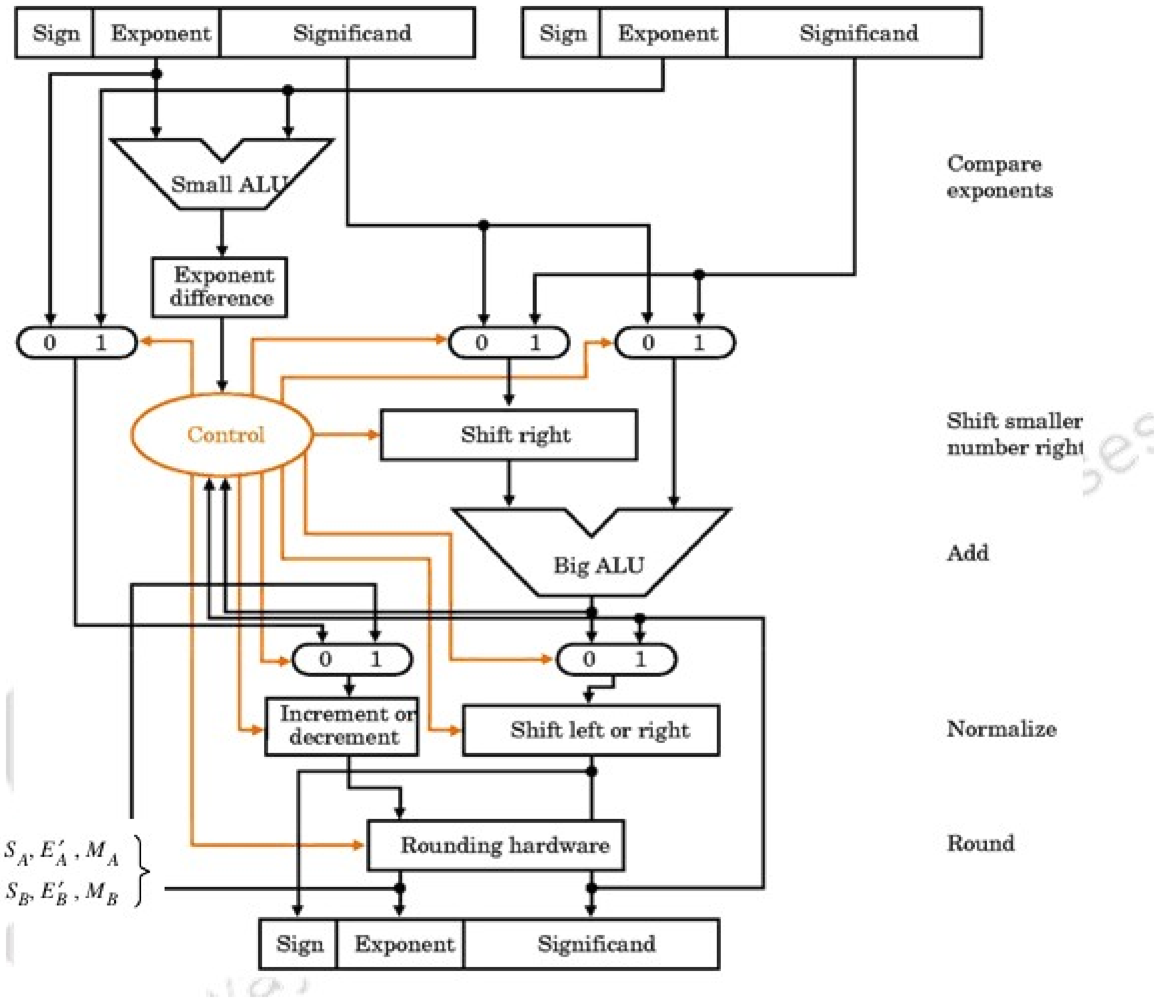




FPU

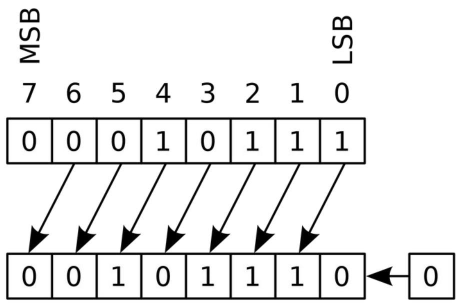


E'_A
E'_B
M_A
M_B
8-bit subtractor
SWAP
M of number with smaller E'
M of number with larger E'
SHIFTER
n bits to right
sign
$n = |E'_A - E'_B|$
S_A S_B
Add/ Subtract
Combinational CONTROL network
Add/Sub
Mantissa adder/subtractor
Sign
Leading zeros detector
Magnitude M
E'_A
E'_B
MUX
E'
X
Normalize and round
8-bit subtractor
$E' - X$
$R : S_R$
E'_R
M_R
32-bit result $R = A + B$
32-bit operands A : S_A, E'_A, M_A B : S_B, E'_B, M_B

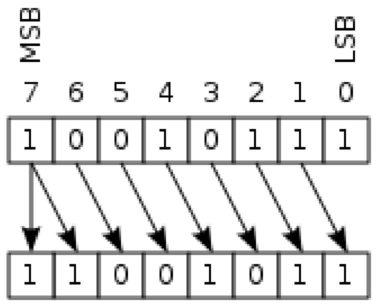


Sign
Exponent
Significand
Sign
Exponent
Significand
Compare exponents
Small ALU
Exponent difference
0 1
0 1
0 1
Control
Shift right
Shift smaller number right
Big ALU
Add
0 1
0 1
Increment or decrement
Shift left or right
Normalize
Rounding hardware
Round
Sign
Exponent
Significand

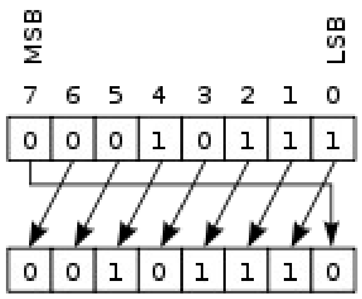
Shifter Functions



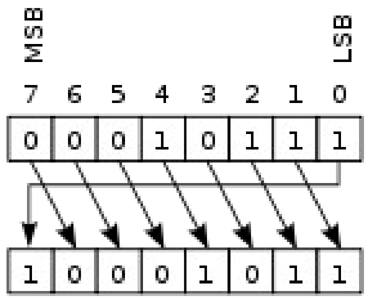
Arithmetic shift left



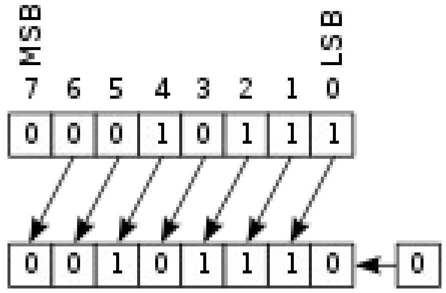
Arithmetic shift right



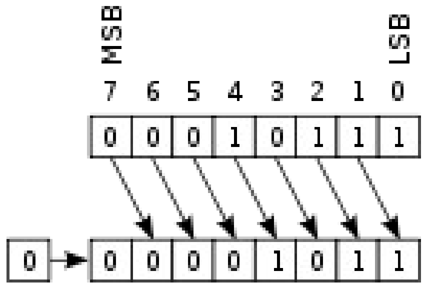
Left circular shift or rotate



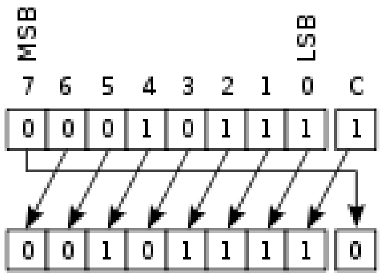
Right circular shift or rotate



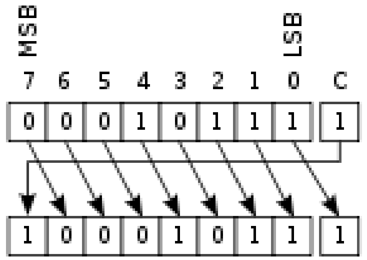
Left logical shift



Right logical shift



Left rotate through carry

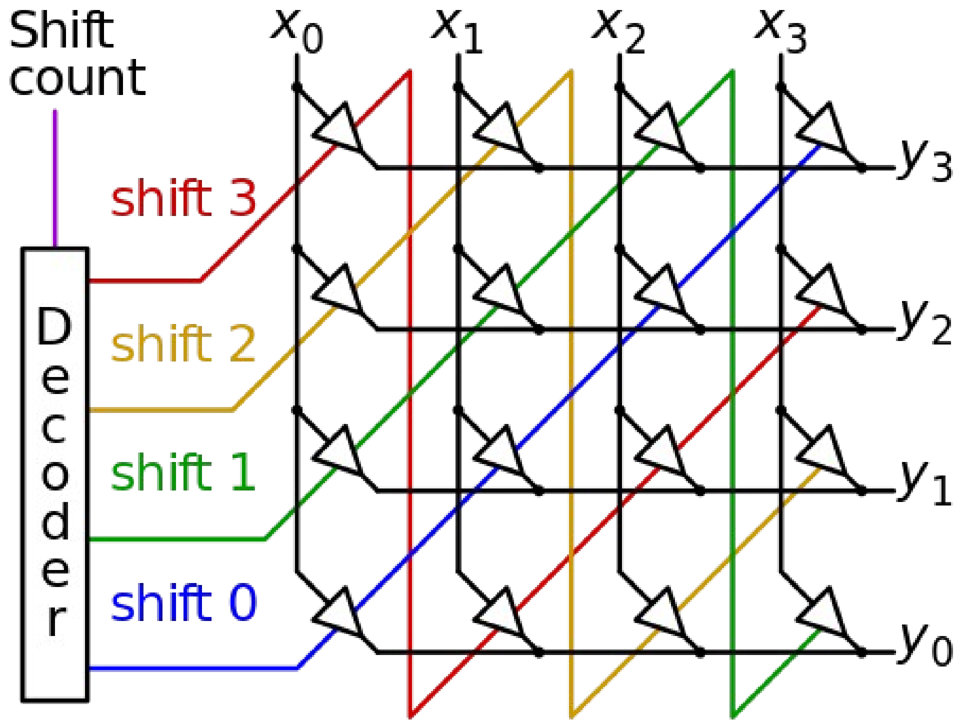


Right rotate through carry

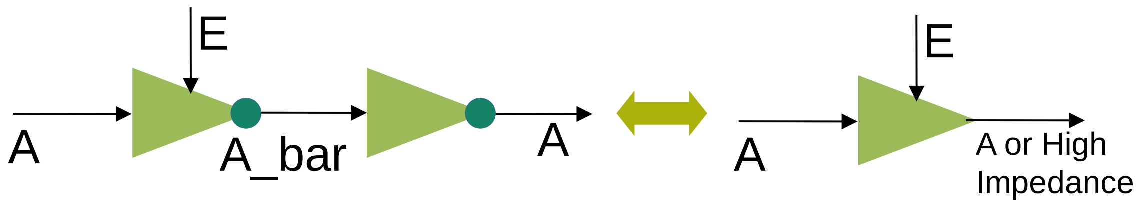
Chandra, Shekhar S. "Fast Digital Convolutions using Bit-Shifts." arXiv preprint arXiv:1005.1497 (2010)

Barrel Shifter

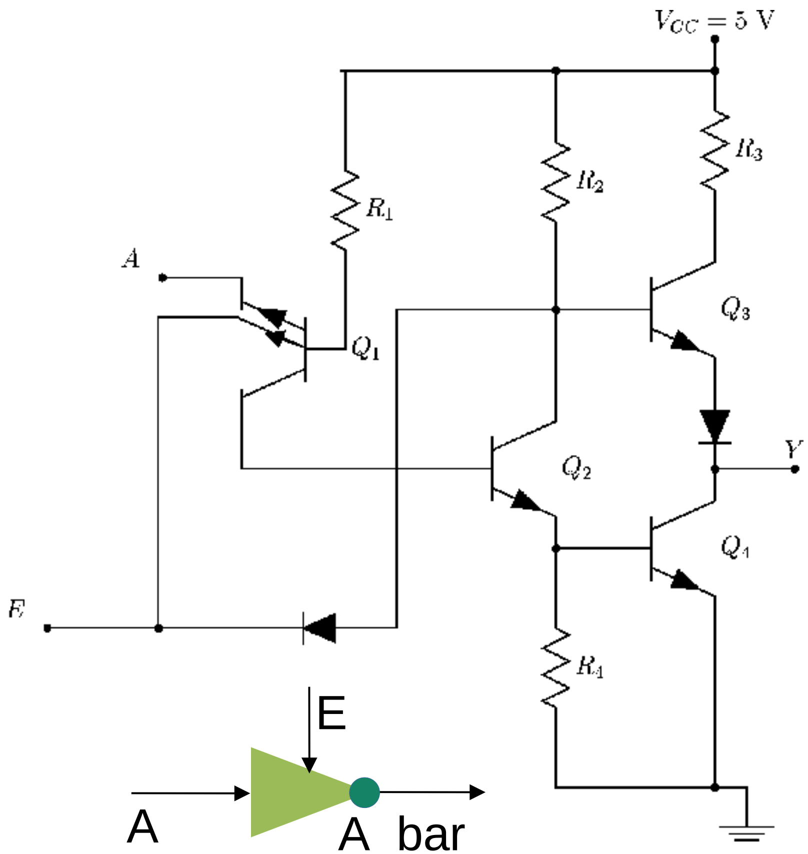
- Barrel Shifter: Def: can shift a data word by a specified number of bits without the use of any sequential logic, but using only pure combinational logic,
  - Tristate GATE controlled by decoder output
  - Is it equivalent to ‘AND’ing the input and decoder output?
  - Can it be implemented by AND?
- Shifting (of any number of bits)
  - Happens in one clock cycle
- Often used to shift and rotate n-bits in a modern RISC based microprocessor (alongside ALU), within a single clock cycle
- A common usage of a barrel shifter is the hardware implementation of floating-point arithmetic
  - Significands of the two numbers must be aligned which requires shifting the smaller number to the right, increasing its exponent, until it matches the exponent of the larger number.
  - The above is done by subtracting the exponents and using the barrel shifter to shift the smaller number to the right by the difference, in one cycle.
  - If the FF based shifter were used, shifting by n bit positions would require n clock cycles.



Barrel shifter model



Tristate GATE



- Relation between size of the data bus and the size of address bus
  - Some designs use the thumb rule that the address bus size is same as data bus size, since addresses could occupy the same space as data
  - Register direct addressing scheme
  - Not always true ---> many designs dont follow this.
- B

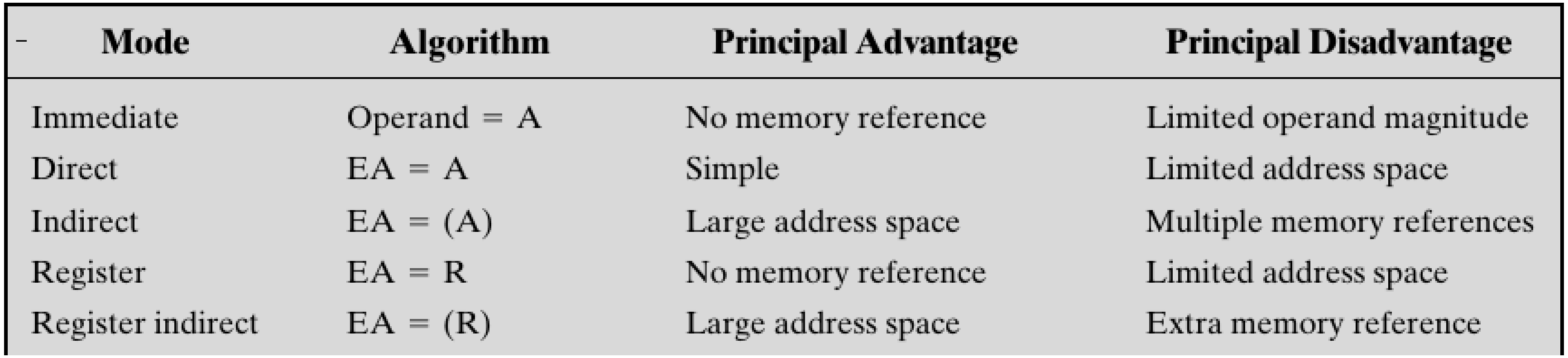
Tristate Inverter GATE (TIG)

Addressing Modes

- Addressing Mode
  - Def: operands have to be supplied along with OP-codes for instruction to be executed
  - to be able to reference a large range of locations in main memory
  - all involve some trade-off between address range and/or addressing flexibility, on the one hand & the number of memory references in the instruction and/or the complexity of address calculation, on the other
- Types of Addressing Modes
  - Immediate
  - Direct
  - Indirect
  - Register Direct
  - Register Indirect

Pros & Cons of Various Addressing Modes

- How the processor decides which Addressing Mode to use?
  - The content in the mode field decides type of addressing mode to use (mode field – or argument of OP-code - is the field which after the OP code in an instruction – for most of the instructions, this field would be non-trivial, meaning the instructions need operand(s))
  - Many instructions also restrict the various types of addressing modes to be used.
- Concept of Effective address (EA)
  - (In designs, without virtual memory) EA is an address of, either, a location in the main memory or, a register in internal memory
  - In virtual memory case, memory management unit (MMU) executes the mapping & is invisible to the programmer
- S

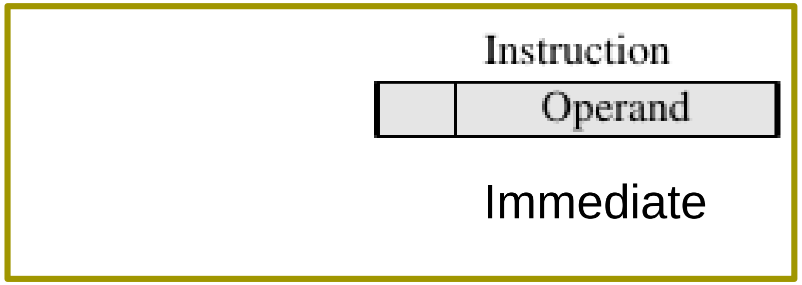


Mode	Algorithm	Principal Advantage	Principal Disadvantage
Immediate	Operand = A	No memory reference	Limited operand magnitude
Direct	EA = A	Simple	Limited address space
Indirect	EA = (A)	Large address space	Multiple memory references
Register	EA = R	No memory reference	Limited address space
Register indirect	EA = (R)	Large address space	Extra memory reference

Addressing Mode: Immediate

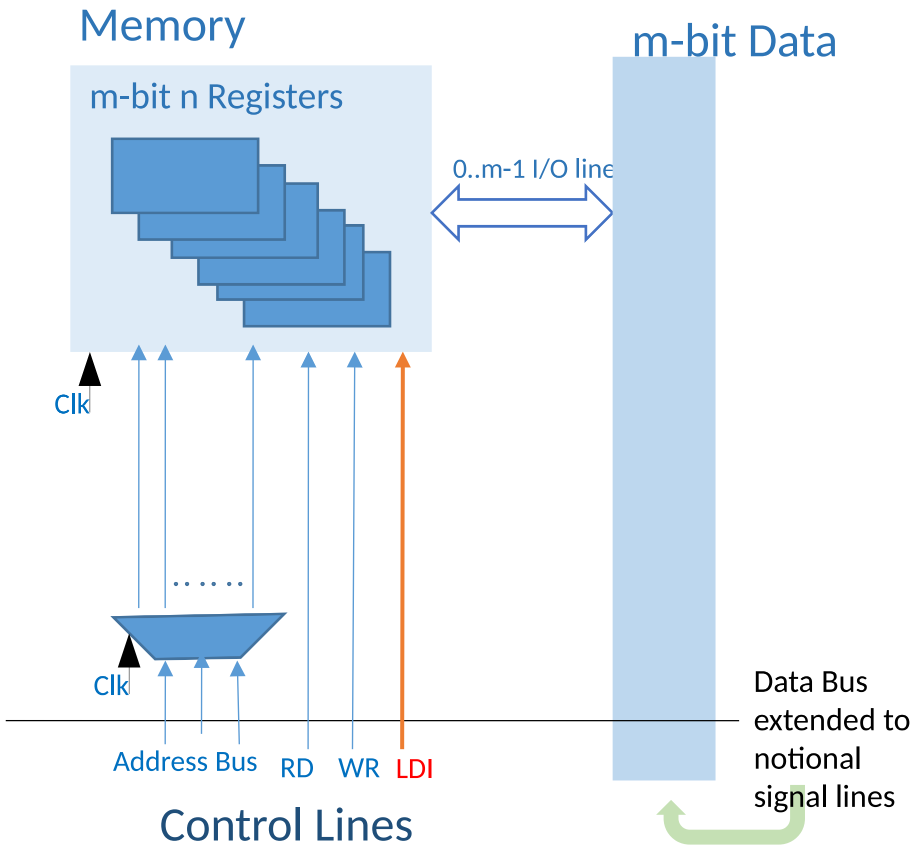
- Immediate Mode `Operand = A`
  - Word in the mode field itself is the operand
  - Advantages
    - No referencing (no address & hence no address computation)
    - Faster memory access (access time less)
  - Disadvantages
    - Operand size is limited to the size of the mode field
    - Range of operands is again limited.
- Example in AVR:
  - `LDI Rd, K;` The 2nd operand follows

    ; immediate addressing

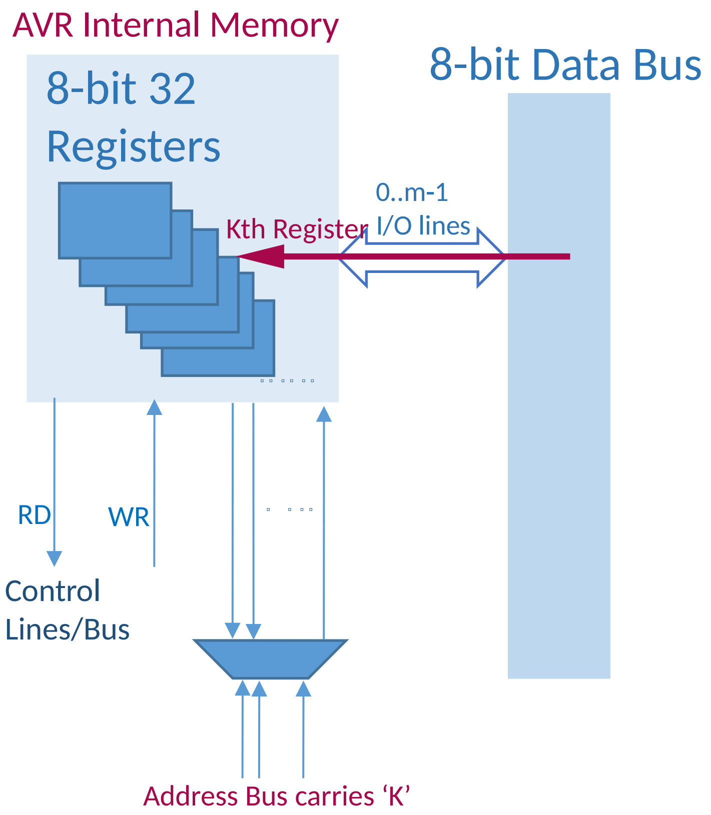


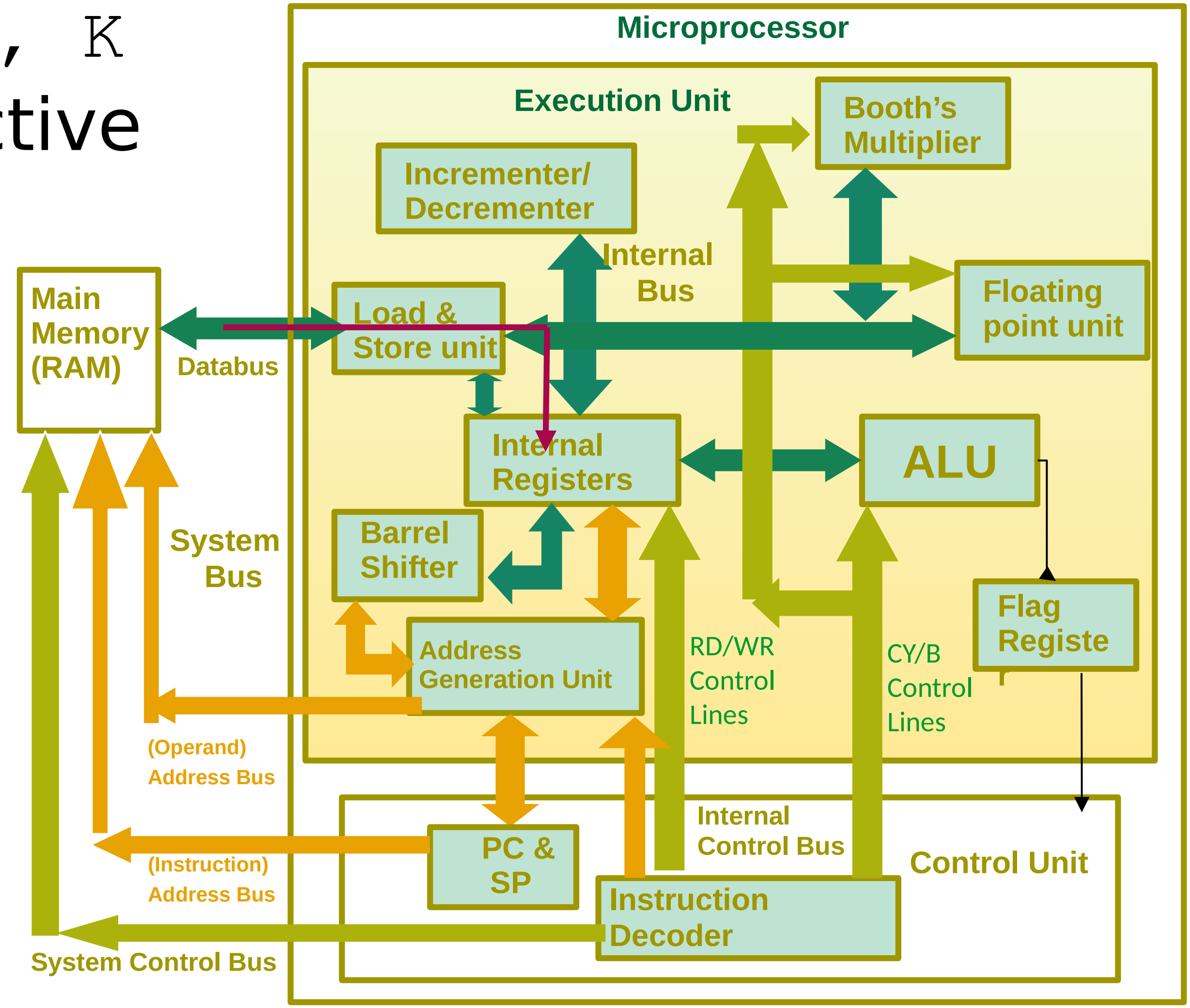
Second Assembly Instruction: LDI

- Load Register (LDI – Load Immediate) enables directly loading values into the registers in a program
  - earlier only the instruction MOV, copying values from other registers was discussed
- To implement LDI,
  - We extend the Data Bus to Notional Signal lines, which means through the external system bus, the internal data bus could be written, a data word externally say, through key board
  - Introduce a new control signal LDI, when it is asserted, the data bits are written into the intended register whose address is held by address bus
- Example Load Instruction
  - LDI R1, 32
    - We put value 32 on the data bus (through say KB) and then read that value from data bus & write into R1
  - The format of the instruction here is LDI <destination>, <value>
    - Some processors might use the format LD <value>, <destination>



Understanding `LDI Rd, K` from Hardware Perspective





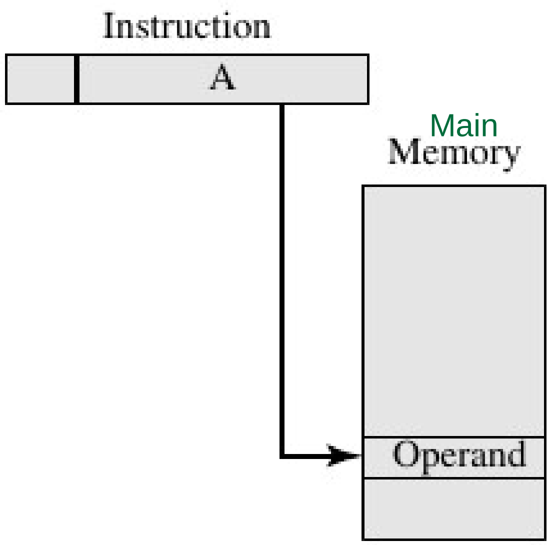
Addressing Mode: Direct

- Direct Mode `EA = A`
  - the mode field contains the effective address of the operand
  - Address field is usually less than the operand word length
  - Advantages
    - Relatively faster (as compared to Indirect mode)
    - Range of operands larger as compared to immediate addressing
  - Disadvantages
    - One memory referencing
  - Example in AVR:
  - `LDS Rd, K;` 0<=d<=31, 0<=K<=65535

    ; Second operand is obtained by direct addressing

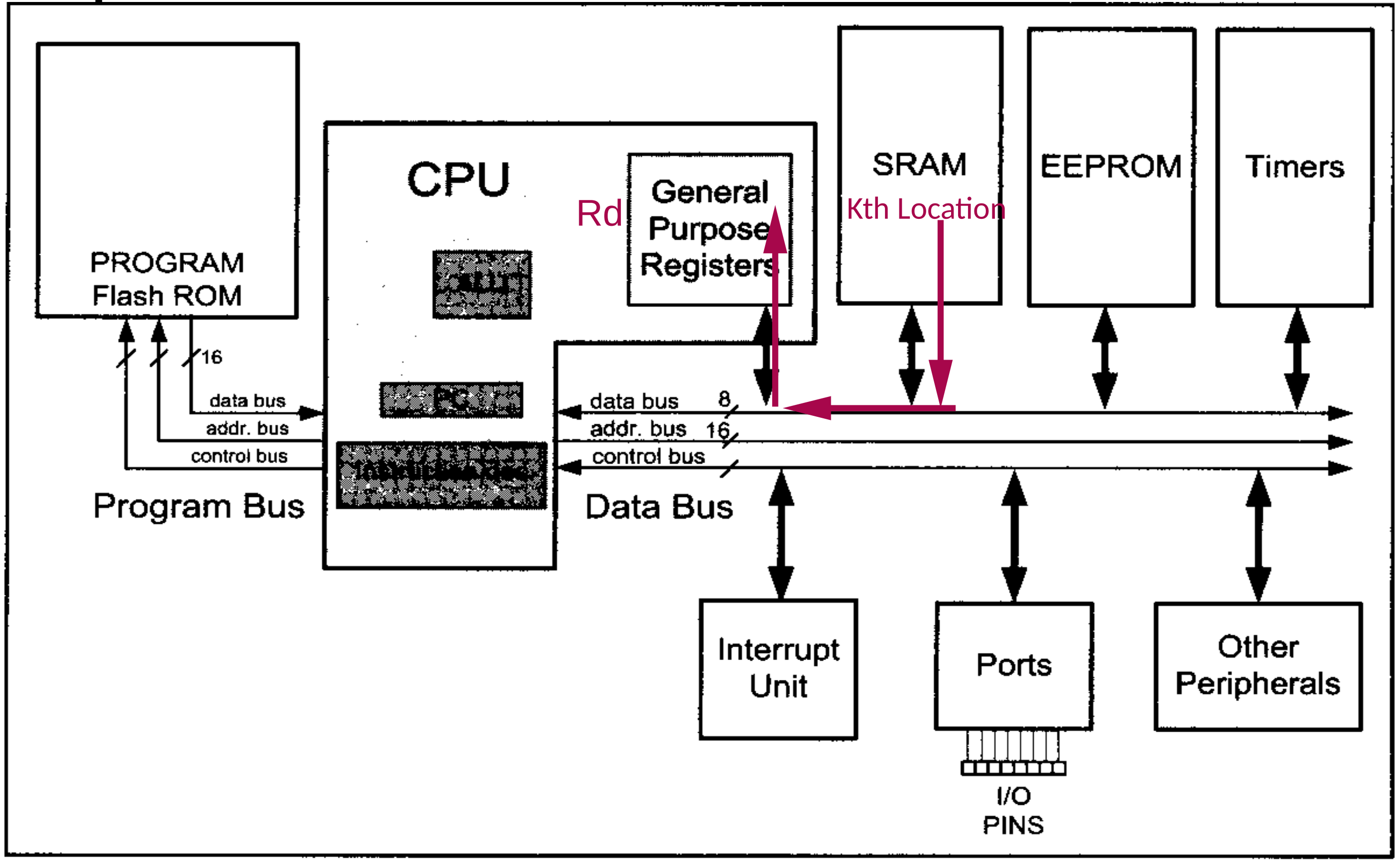
    ;Load direct from data space, Rd = (K). Loads one

    ;byte direct from data space (external SRAM)
- So also `STS K, Rs;`



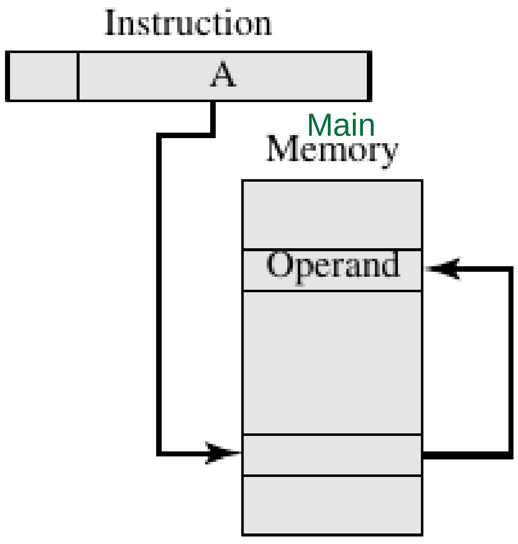
Direct

Understanding `LDS Rd, K` from Hardware Perspective



Addressing Mode: Indirect

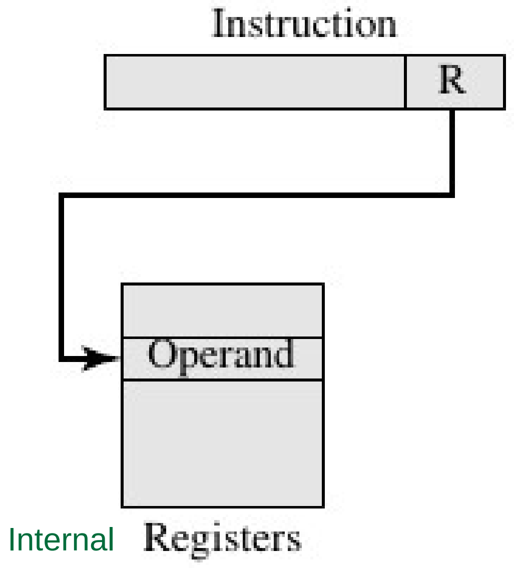
- Indirect Mode `EA = (A)`
  - Effective address is the address given by the value in the address location denoted by A
  - Advantages
    - Range of operands larger
  - Disadvantages
    - 2 memory references (higher computational overhead)
  - `Example ARM:`



Indirect

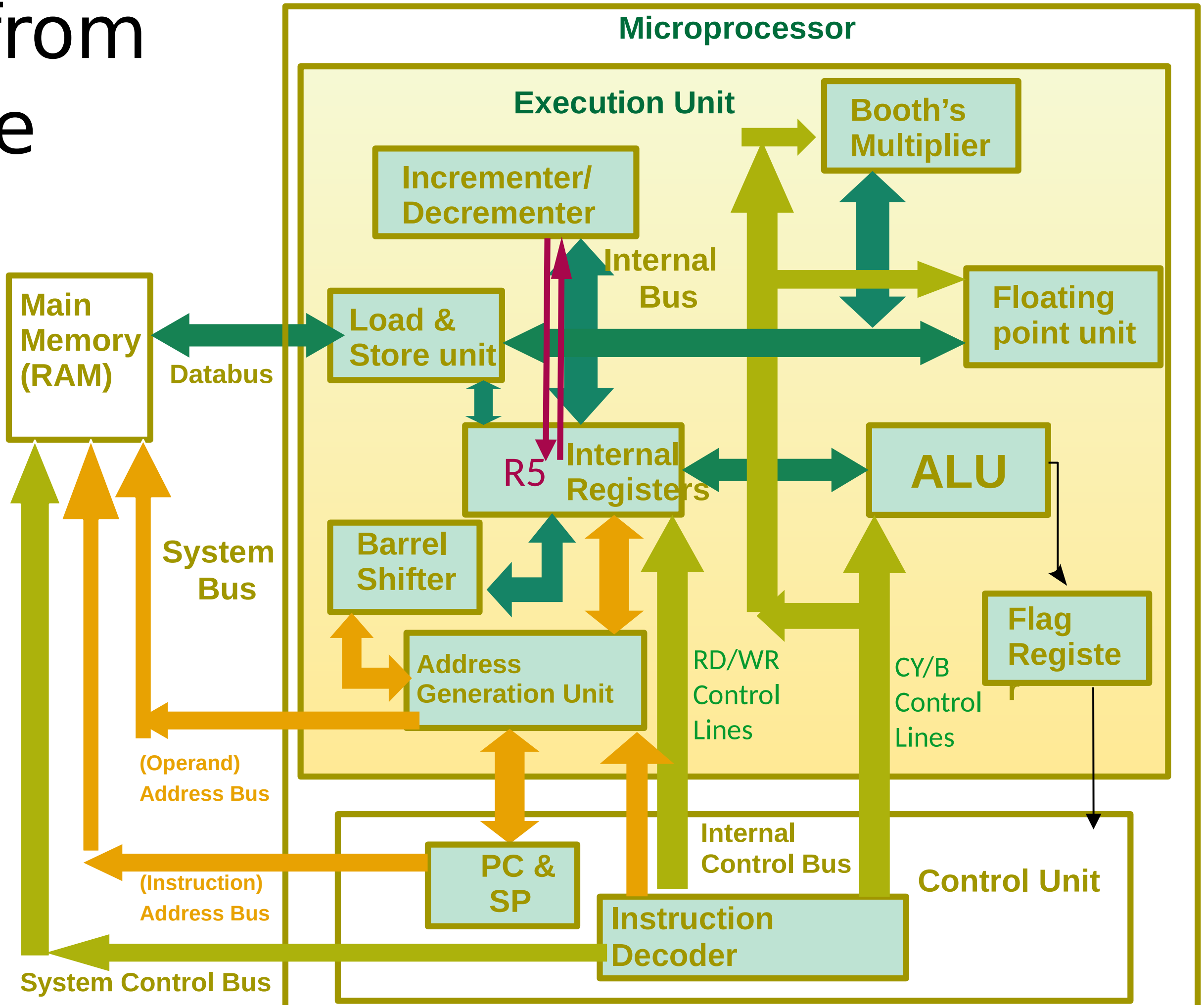
Addressing Mode: Register Direct

- Register Direct `EA = R`
  - Address is in register (internal) number.
  - The operand is the content of the specific register.
  - Advantages
    - Small address field
    - Simple & faster memory referencing (access time for internal registers is less)
  - Disadvantage
    - Less address space
  - Used for the intermediate results by EU
  - Eg: `INC R5`
- .



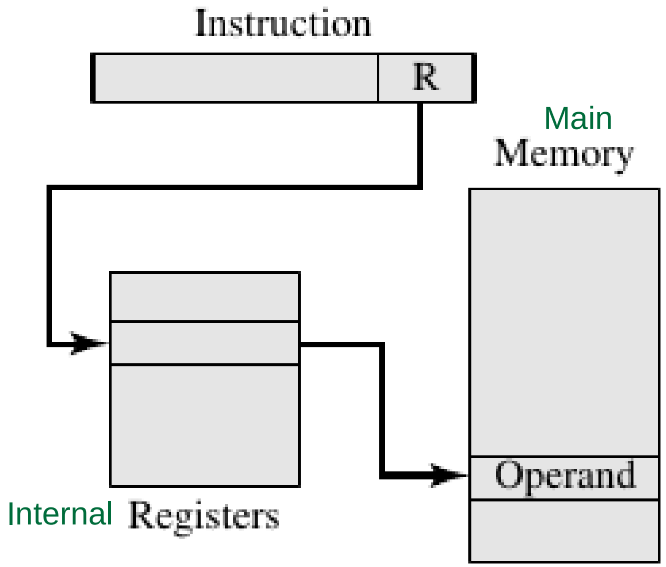
Register Direct

Understanding `INC R5` from Hardware Perspective



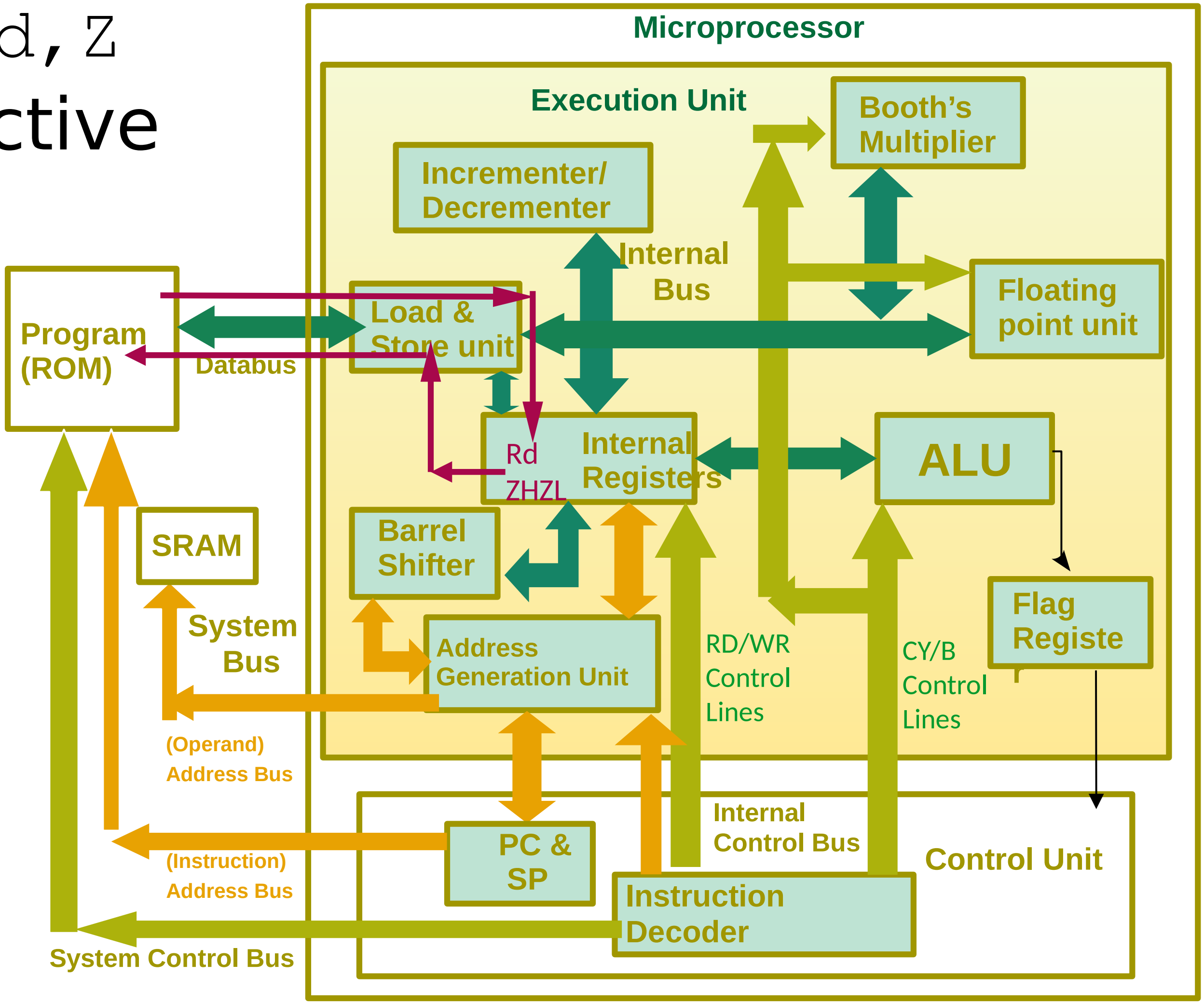
Addressing Mode: Register Indirect

- Register Indirect `EA = (R)`
  - Address in specified register
  - Operand can be instruction or data
  - In AVR, X or Y or Z registers used as pointers (to program ROM or data space)
- Eg: `LPM R1, Z`
  - 2nd operand is register indirect
  - Advantages
    - Memory locations addressed is large
  - Disadvantage
    - Double memory reference
  - Used by EU to store the final result or used by control unit to pick the instruction
- Eg: `LPM Rd, Z;` Load program memory. Load one byte pointed to, by Z-register into the destination GPR Rd. In AVR, instructions (in program ROM) is two bytes, while Z holds the byte address. Used to pick the semi-permanent data in the program ROM.



Register Indirect

Understanding `LPM Rd,Z` from Hardware Perspective

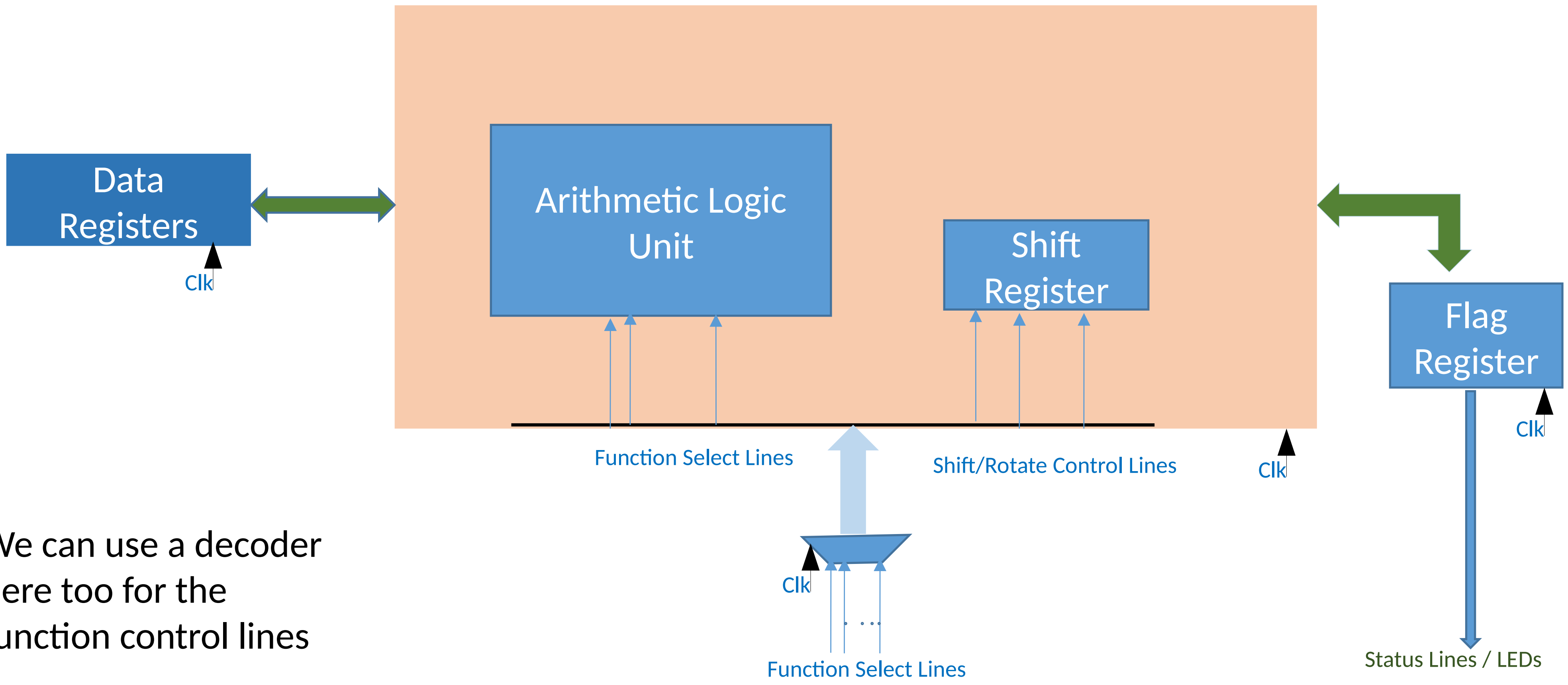


Addressing Modes & Range of Operands

- Each mode field is 8 bits
- Internal memory 32, 8-bit registers
- 1 kbyte of main memory with word size 2 bytes

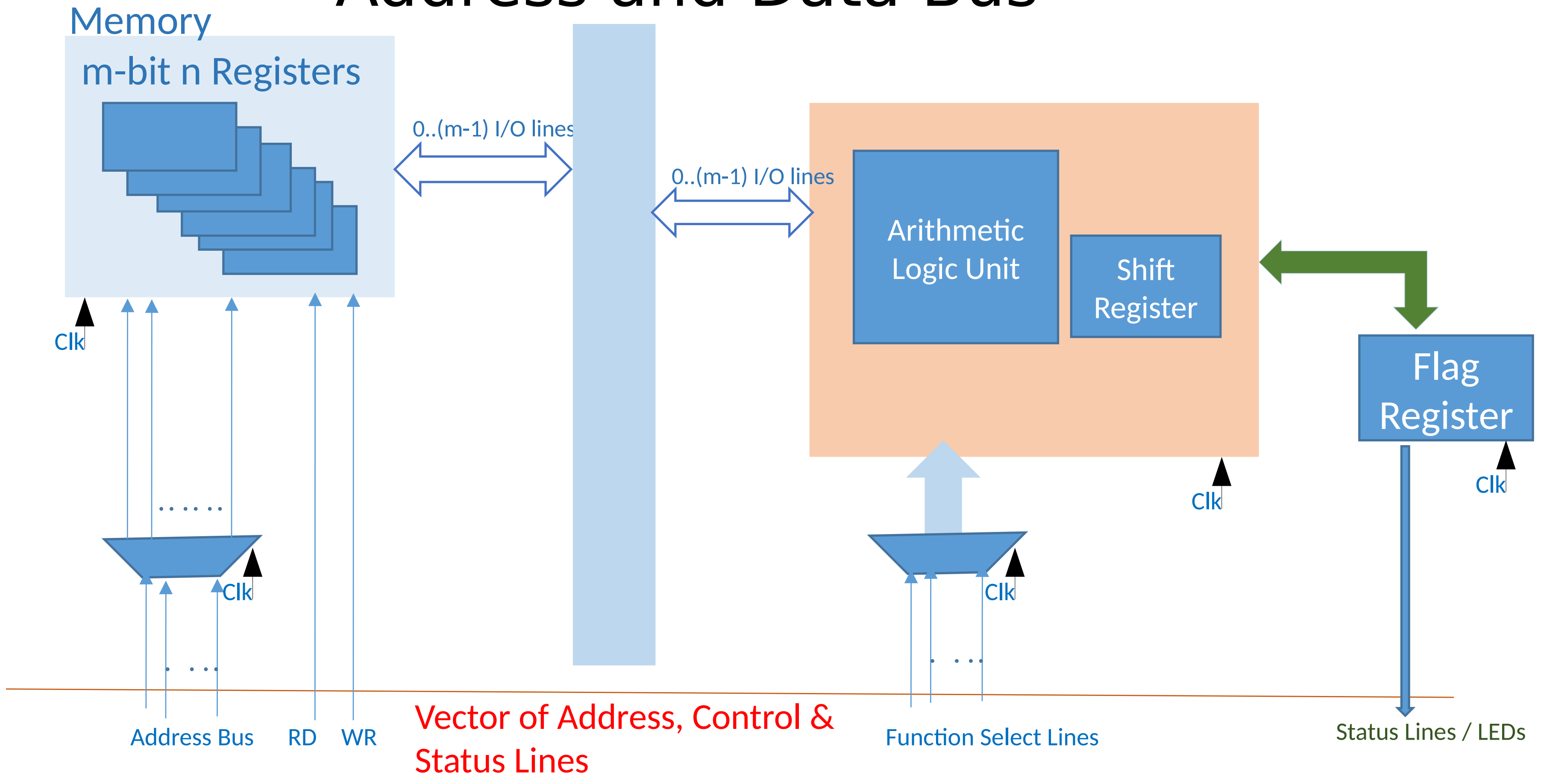
S. No	Type of Addressing	Mode field size	How many different unique memory (final) locations (containing operands), it can point to?	Range of operand
1	Immediate	8	NIL – no memory reference	2^8
2	Direct	8	2^8 main memory (mm) locations (unique mm* words of 2 byte length)	2^16
3	Indirect	8	2^8 addresses (mm) which in turn CAN point to 2^16 addresses. But, here points to 2^9-1 mm locations each containing mm words of 2 byte length	2^16
4	Direct register	5	2^5 (im*) locations (registers) each of size 1 byte	2^8
5	Indirect register	5	2^5 (im*) addresses, whose value (1 byte address) pointing to 2^8 mm locations containing 2 bytes word	2^16
			mm* - main memory, im* - internal memory	

Control Lines for ALU and Shifter



We can use a decoder here too for the function control lines

ALU with SR, Registers interconnected with Control, Address and Data Bus



Programming using Machine Language Instructions

We have learned design of EU so far

Now let us learn how to drive the EU as per a program: And this is manual (rather than automatic).

Translating Assembly Language Instructions to Control Signals

- An assembly program
  - pertains to an implementation of a given task
  - consists of a sequence of instructions.
  - And each instruction consists of two main fields: an OP-CODE or command and operands
  - First we would manually pick each instruction & execute (without the feedback loop in fetch, execute loop)
- Execution of an Assembly language instruction may have multiple steps
  - Trigger a state machine for each step
  - These state machines work sequentially in time

Instruction	Remarks	Control Signals for State Machine
MOV	A 2 Step instruction (In AVR, 1 cycle)	1. Set address bus to source reg, press READ 2. Set address bus to destination register, press WRITE
LDI	Need to first load a value in data bus and then perform write on destination register	1. Set data bus with the value, press LOAD 2. Set address bus to destination, Press WRITE
ADD R0, R1	1 step instruction. Load operands & then select the corresponding function select line	Select the corresponding function select line

Example: Simplest Microcontroller – AVR ATmega8

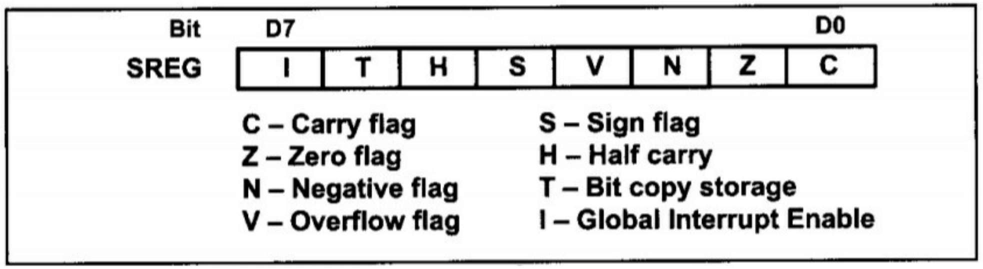
- Thirty-two 8 bit-Registers
  - 5 Address Lines (A4, A3, A2, A1, A0)
- 8 Bit Data Lines
- 4 Bit Flag Register (hypothetical)
- Hypothetical (not in AVR) - 4 function ALU with
  - R0, R1 Mapped input for ADD and SUBTRACT
  - R4, R5 Mapped input for COMPARE
  - R0 Mapped input for INC
  - R2 Mapped Output
  - 2 ALU function select lines F1, F0

ALU Functions

Function Select Line	Function	Code
0	ADD	ADD
1	COMPARE	CP
2	SUBTRACT	SUB
3	INCREMENT	INC

Flag Register (hypothetical)

Bit no	Flag name	Description
0	L0	Carry/Borrow/ Overflow/ underflow
1	L1	Shifted out bit
2	L2	is set if any ALU operation results in Zero
3	L3	1 indicates result of compare is true, 0 otherwise


Bit D7 D0

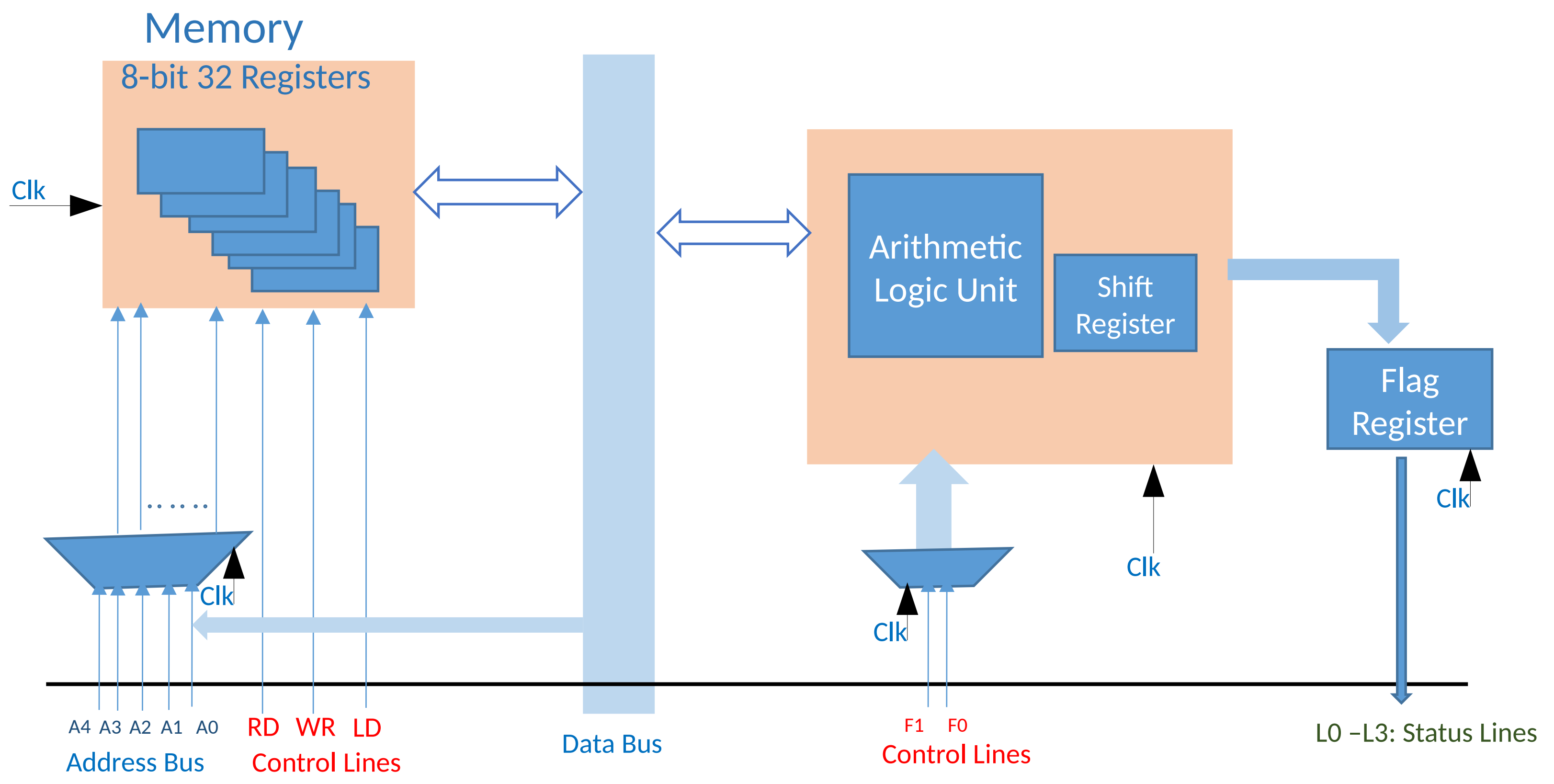
SREG

I	T	H	S	V	N	Z	C

C – Carry flag
Z – Zero flag
N – Negative flag
V – Overflow flag
S – Sign flag
H – Half carry
T – Bit copy storage
I – Global Interrupt Enable

SREG Flags in AVR

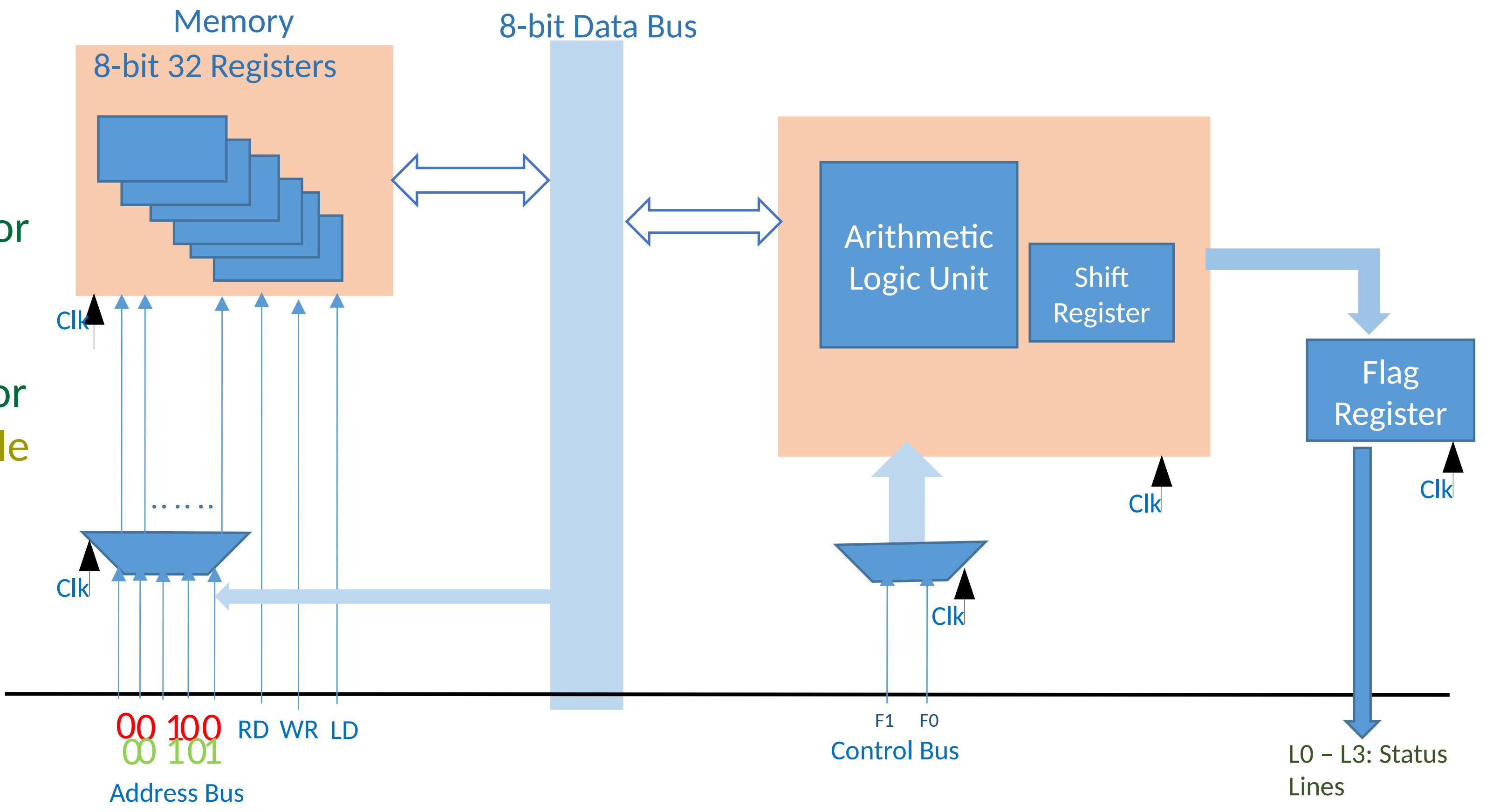
Execution Unit Model



Implement Instruction: MOV R5, R4

Set Address Bus for Source, Select RD

Set Address Bus for Destination; Enable WR



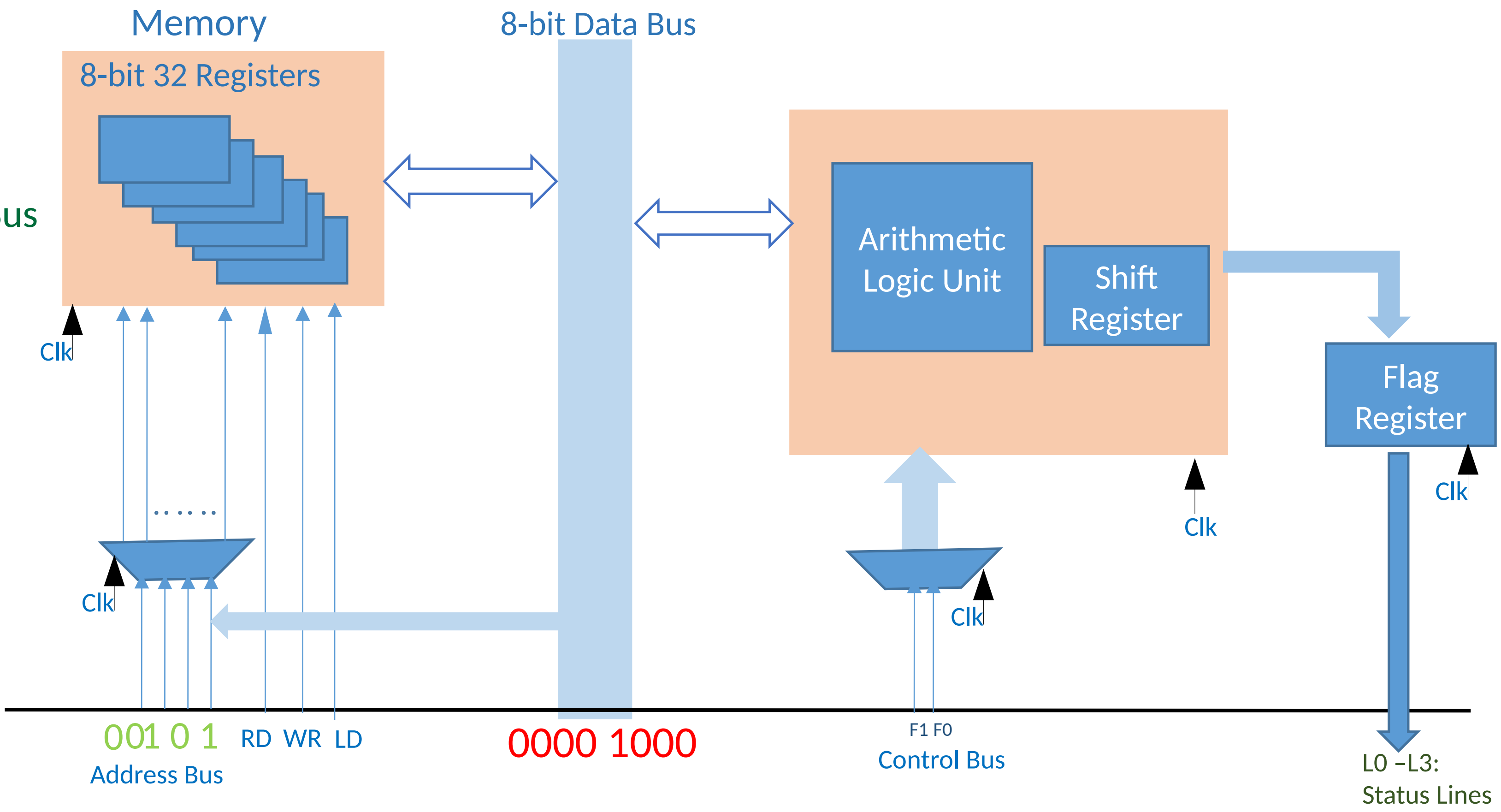
Now Implement Instruction: LOAD 8, R5

Set value 8 in Data Bus

Select LD

Set 5 in address bus

Enable WR



Program: Load 2 values, add them and store the carry in R5

Load 2 values as 40 and 50 in R0 and R1

Instructions
LDI R0, 40
LDI R1, 50
ADD R0, R1
BRCS LOOP
LDI R5, 0
NOP
LOOP: LDI R5, 1
NOP

Add the two values to check for carry

=

Data Bus	A4	A3	A2	A1	A0	RD	WR	LD	IA	C1	C0	Flag
0010 1000	X	X	X	X	X	0	0	1	0	X	X	0000
0010 1000	0	0	0	0	0	0	1	0	0	X	X	0000
0011 0010	X	X	X	X	X	0	0	1	0	X	X	0000
0011 0010	0	0	0	0	1	0	1	0	0	X	X	0000
xxxx xxxx	X	X	X	X	X	X	X	X	X	0	0	0000
xxxx xxxx	X	X	X	X	X	X	X	X	X	X	X	0000

Carry Generated

Data Bus	A4	A3	A2	A1	A0	RD	WR	LD	IA	C1	C0	Flag
0000 0001	X	X	X	X	X	0	0	1	0	X	X	0001
0000 0001	0	0	1	0	1	0	1	0	0	X	X	0001

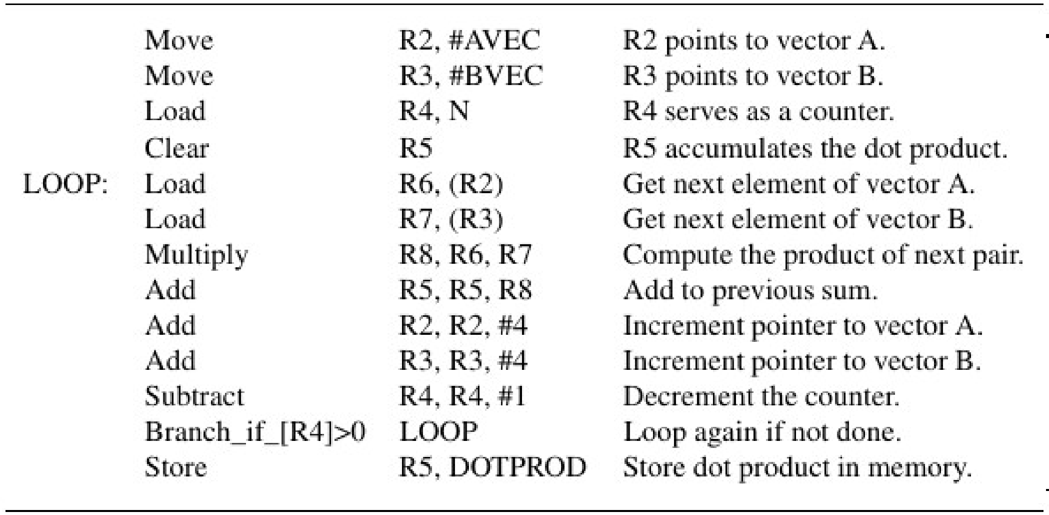
No Carry Generated

Data Bus	A4	A3	A2	A1	A0	RD	WR	LD	IA	C1	C0	Flag
0000 0000	X	X	X	X	X	0	0	1	0	X	X	0000
0000 0000	0	0	1	0	1	0	1	0	0	X	X	0000

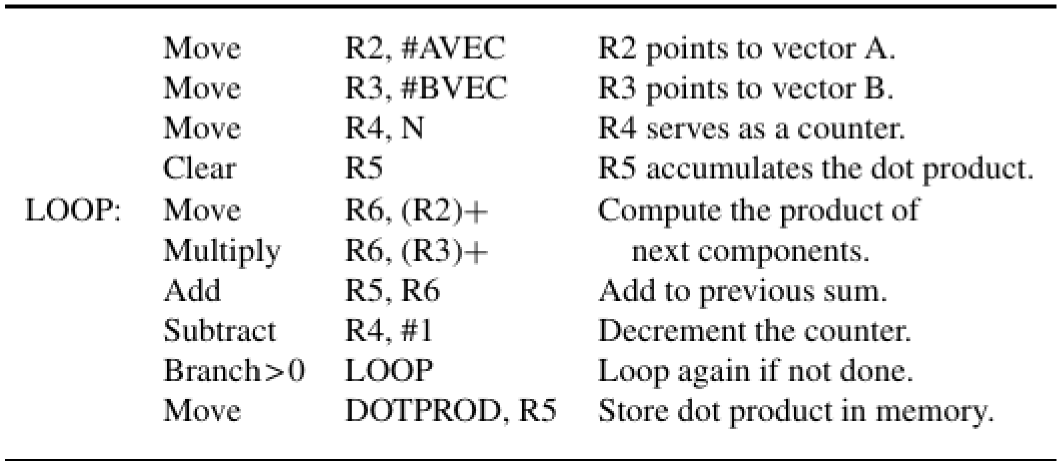
To sum up: Assembly Program (Manual)

- Each instruction in the (assembly) program here is taken in sequence (or jump according to previous instruction) manually.
- This is automatically done by the Program Counter (PC) and conditional / unconditional jumps alter the sequence which is again tracked my PC.
- Thus, PC makes the whole program execution automatic (external manual intervention is not required).

RISC Style Assembly Program versus CISC



```
        Move              R2, #AVEC      R2 points to vector A.
        Move              R3, #BVEC      R3 points to vector B.
        Load              R4, N          R4 serves as a counter.
        Clear             R5             R5 accumulates the dot product.
LOOP:   Load              R6, (R2)       Get next element of vector A.
        Load              R7, (R3)       Get next element of vector B.
        Multiply          R8, R6, R7     Compute the product of next pair.
        Add               R5, R5, R8     Add to previous sum.
        Add               R2, R2, #4     Increment pointer to vector A.
        Add               R3, R3, #4     Increment pointer to vector B.
        Subtract          R4, R4, #1     Decrement the counter.
        Branch_if_[R4]>0  LOOP           Loop again if not done.
        Store             R5, DOTPROD    Store dot product in memory.
```



```
        Move        R2, #AVEC      R2 points to vector A.
        Move        R3, #BVEC      R3 points to vector B.
        Move        R4, N          R4 serves as a counter.
        Clear       R5             R5 accumulates the dot product.
LOOP:   Move        R6, (R2)+      Compute the product of
        Multiply    R6, (R3)+        next components.
        Add         R5, R6         Add to previous sum.
        Subtract    R4, #1         Decrement the counter.
        Branch>0    LOOP           Loop again if not done.
        Move        DOTPROD, R5    Store dot product in memory.
```

Which of them is RISC and which of them is CISC? Mention the line(s) which identifies the respective style. Reason it out.

You have so far learned

- How to design EU of a processor?
- How to drive EU using Control Signals?
- Driving Control Signals as per a program
  - Usually instructions are executed in sequence
  - Occasionally change sequence (program flow) as per the outcome of the current instruction(s)
- Next we will learn
  - How to automate the Controller (so far a human being was a controller – or we just supplied control signals, did not specify how they were generated).
  - How the above is achieved using a Control Unit and Program Memory?

Applications for Assembly Language

- Theoretical reasons
  - assembly programming can have tighter control over the hardware, upon execution.
  - Given an engineering task, optimizations could be done in terms of number of computations, power consumed, computational time, latency, run-time memory utilized, number of logical blocks utilized etc, if one uses assembly programming
- Practical reasons / applications of assembly programming
  - Short to moderate-sized programs
  - High volume applications
  - Applications involving more input / output or control than computation
- .

Assembly Language Relevance in 2024?

- While with C-Interfacing, we could program microcontrollers, why is that we need assembly programming?
  - Theoretical reasons
    - assembly programming can have tighter control over the hardware, upon execution.
    - Given an engineering task, optimizations could be done in terms of number of computations, power consumed, computational time, latency, memory utilized, number of logical blocks utilized etc, if one uses assembly programming
  - Practical reasons
    - Short to moderate-sized programs
- It depends on what one intends to work in
  - For following, assembly programming is usually needed: Web apps, Payroll systems, Windows apps, Almost any GUIs, hardware drivers, BIOS, operating systems, kernel programming, or compilers or program microcontrollers, embedded systems programming or writing device drivers, context switching
  - In area of resource constrained embedded systems
    - Resource constraining means, 4 and 8 bit microcontrollers, with smaller memory, lesser frequency, which may or may not have any C-compilers.
      - These muC are still widely in use, since industry shows no sign of a reduction in 8-bit usage, and their low-cost nature combined with simplicity still makes them highly relevant
      - a consumer product costs less than a dollar, ships millions of units per year. Handles smaller tasks. Every penny you save on materials can turn into tens of thousands of dollars in extra profit. Use assembly programming
    - Bank switching:
      - One of the resource which is constrained in such systems is memory. Bank switching is used to increase the addressable memory by the processor. Eg microchips, PICs
      - Processor with a 16-bit external address bus can only address 2^16 = 65536 memory locations. If an external latch was added to the system, it could be used to control which of two sets of memory devices, each with 65536 addresses, could be accessed.
- Assembly programming to implement bank switching

Assembly language relevance in 2024

- There are some aspects of computer science, where it is not only useful, but pretty much necessary to learn assembly language at some point in your studies:
  - Embedded programming on small microcontrollers — we are talking about 8-bit ones with maybe only a KB or less of flash memory, and a couple of hundred bytes of RAM, if that — such as a PIC10F322.
  - Writing low-level operating system routines, such as task switching where you need access to hardware registers.
  - Writing a back-end (code generator) for a compiler.
  - If you are going to use C or another language that uses pointers, and you want to really understand how they work.
- Besides that, it is not necessary, but would make one understand better, how a computer processor works internally.

Placement Requirements: Assembly Language Expertise & Domain Knowledge

- Employers need the candidate's familiarity with assembly language
  - this helps in identifying and addressing bottlenecks in execution time in time-critical code sections.
  - Familiarity with one or two ISAs is sufficient, the rest can be learnt on the job.
- Given an embedded program in C,
  - identify the sections in the above code, the performance bottlenecks
  - Address these bottlenecks, by replacing the section with corresponding assembly program section devoid of bottlenecks
- Crictical loops part of the code
  - FFT, matrix multiplications, part of the code too often would run --> write it in assembly and other parts write it in C, then link them all to have one unified object code which is the application
  - Low level loops in parallel and distributed computing --> write it in assembly which are close to hardware