

EE XXX

Design of Intelligent SoC Robot

<http://ssl.kaist.ac.kr/class/18fall/socrobotdesign>

Hoi-Jun Yoo

KAIST

Processor and Board



Outline

1. What is a processor?

2. How can we design the processor?

- *Single Purpose Processor Design*
- *Optimization*
- *Architecture-level Design*
- *System-level Design*

3. Brain Board

Outline

1. Introduction to Processor

- What is a processor and its purpose?***

2. How can we design the processor?

- Single purpose processor Design***
- Optimization***
- Architecture-level Design***
- System-level Design***

3. Brain Board

Embedded system functionality aspects

- Processing and Control

- Transformation of data at the right time for the right device and for the right operation
- Implemented using **processors**

- Storage

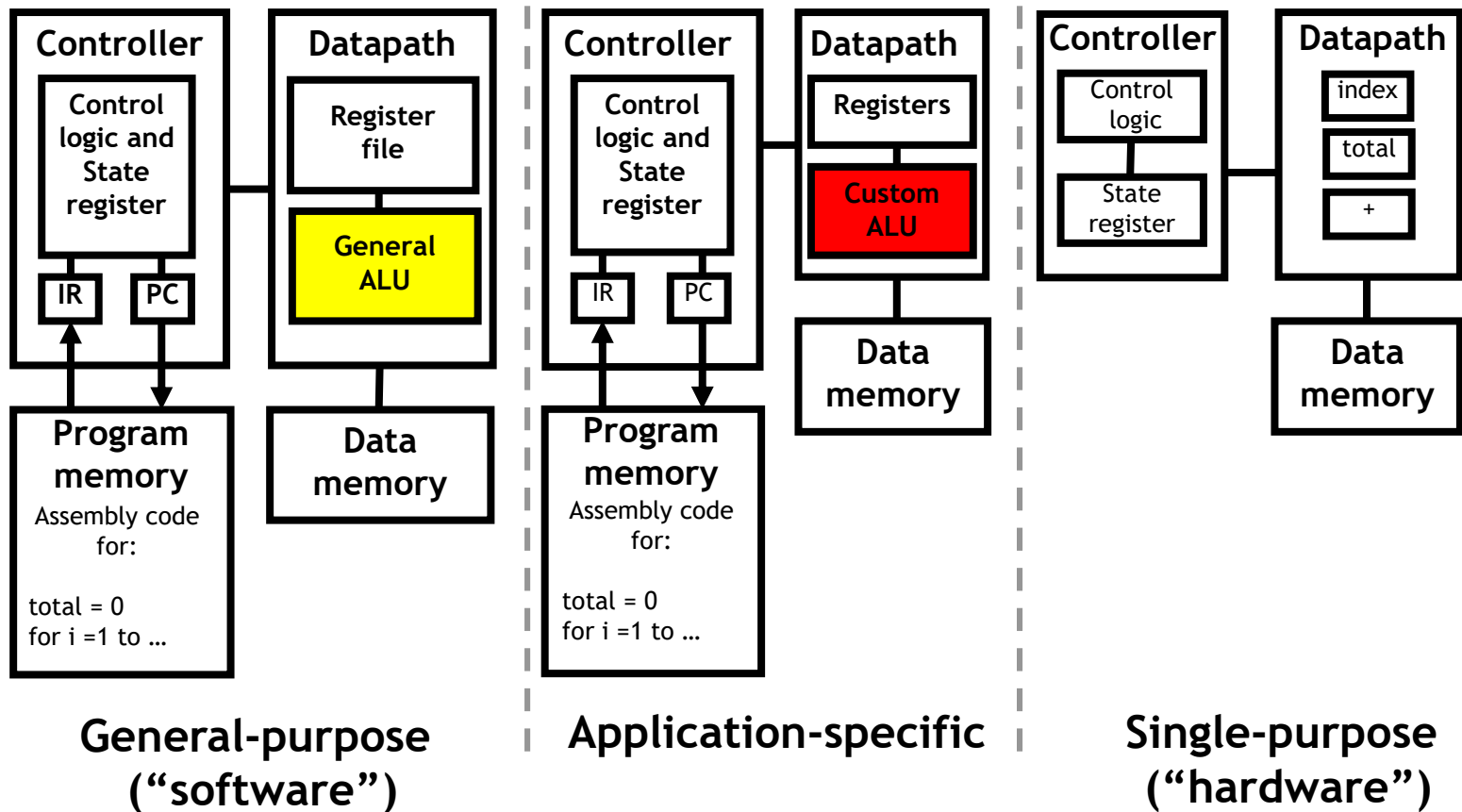
- Retention of data
- Implemented using **memory**

- Communication

- Transfer of data between processors and memories
- Implemented using buses
- Called ***interfacing***

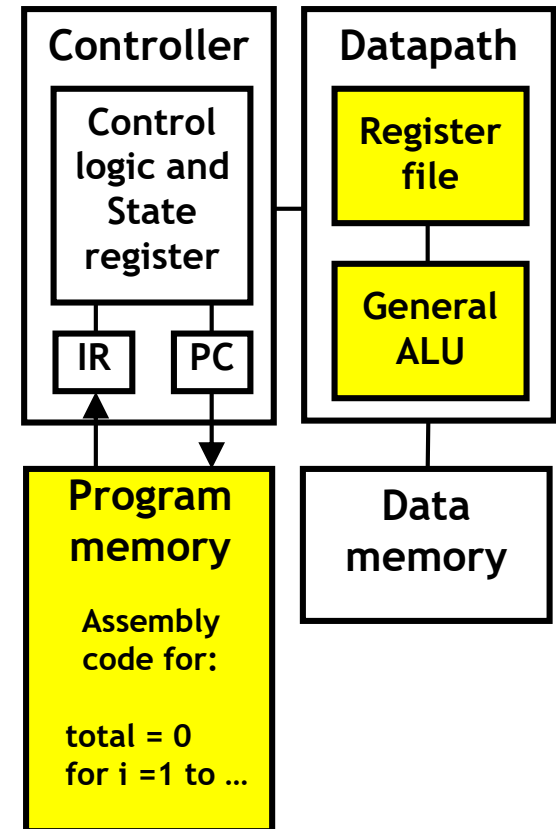
Processors

- The architecture of the computation engine used to implement a system's desired functionality
- **Processor does not have to be programmable**
 - “Processor” *not* equal to general-purpose processor



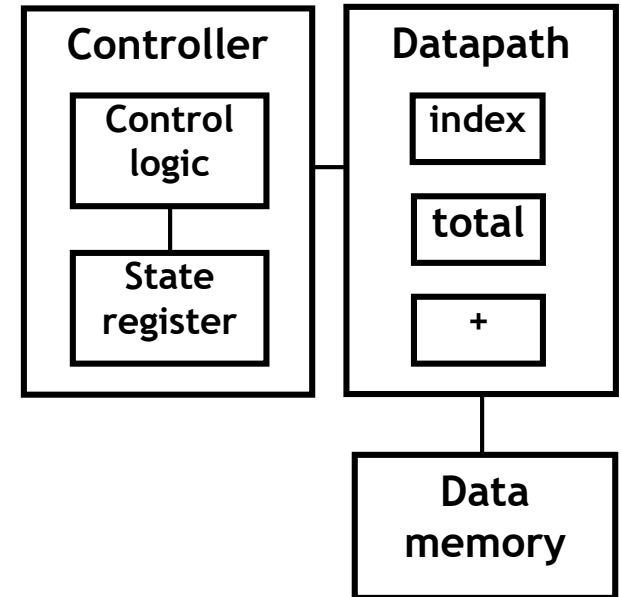
General-purpose processors

- **Programmable** device used in a variety of applications
 - Also known as “microprocessor”
- Features
 - Program memory
 - General datapath with large register file and general ALU
- User benefits
 - Low time-to-market and NRE costs
 - High flexibility
- “ARM” the most well-known, but there are hundreds of others



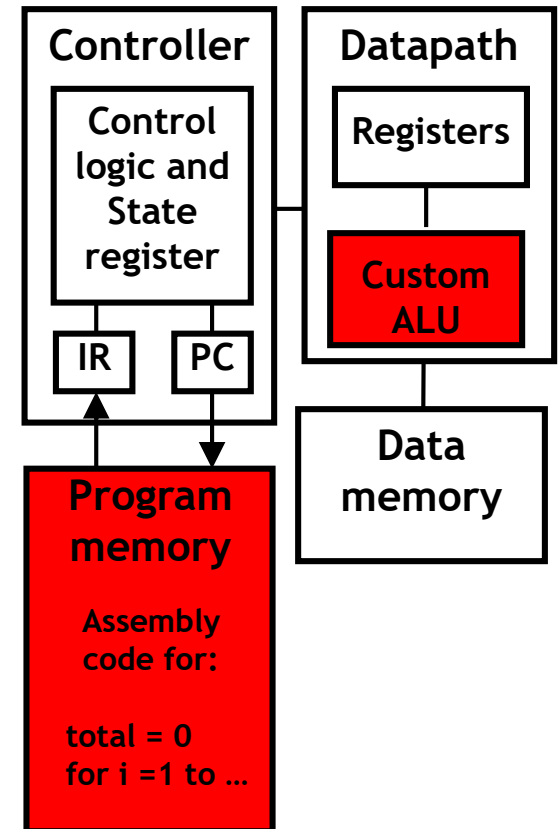
Single-purpose processors

- Digital circuit designed to execute exactly one program
 - a.k.a. coprocessor, accelerator or peripheral, usually with FPGA.
- Features
 - Contains only the components needed to execute a single program
 - **No program memory**
- Benefits
 - Fast
 - Low power
 - Small size



Application-specific processors

- Programmable processor optimized for a particular class of applications having common characteristics
 - Compromise between general-purpose and single-purpose processors
- Features
 - Program memory
 - Optimized datapath
 - Special functional units
- Benefits
 - Some flexibility, good performance, size and power



Outline

1. What is a processor?

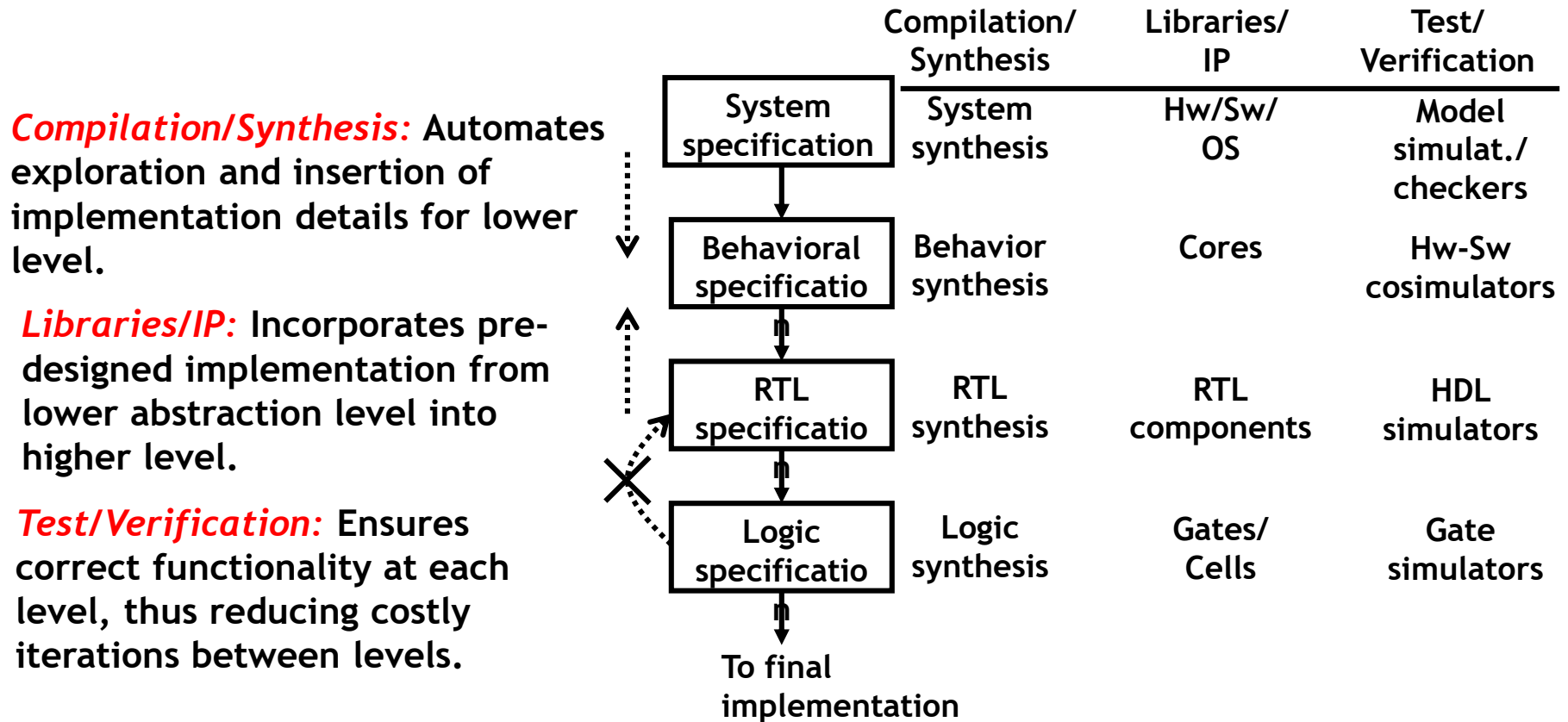
2. How can we design the processor?

- *Single purpose processor Design*
- *Optimization*
- *Architecture-level Design*
- *System-level Design*

3. Brain Board

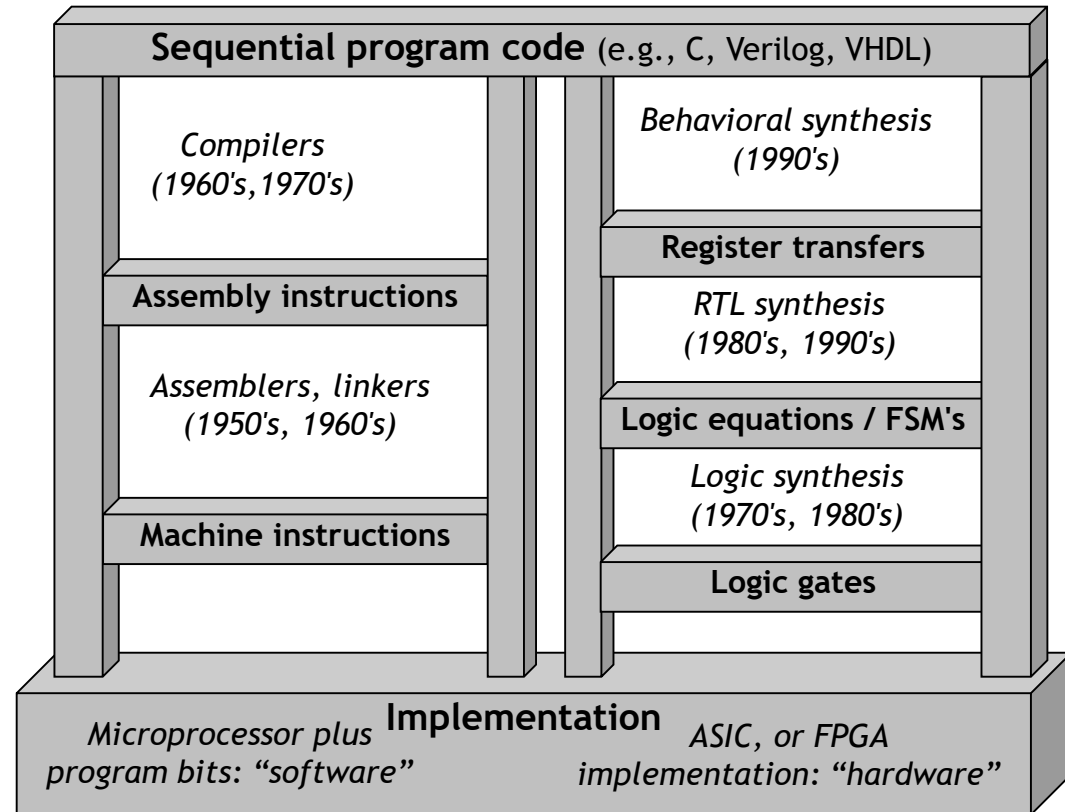
Design Technology

- The manner in which we convert our concept of desired system functionality into an implementation



The co-design ladder

- In the past:
 - Hardware and software design technologies were very different
 - Recent maturation of synthesis enables a unified view of hardware and software
- Hardware/software “codesign”

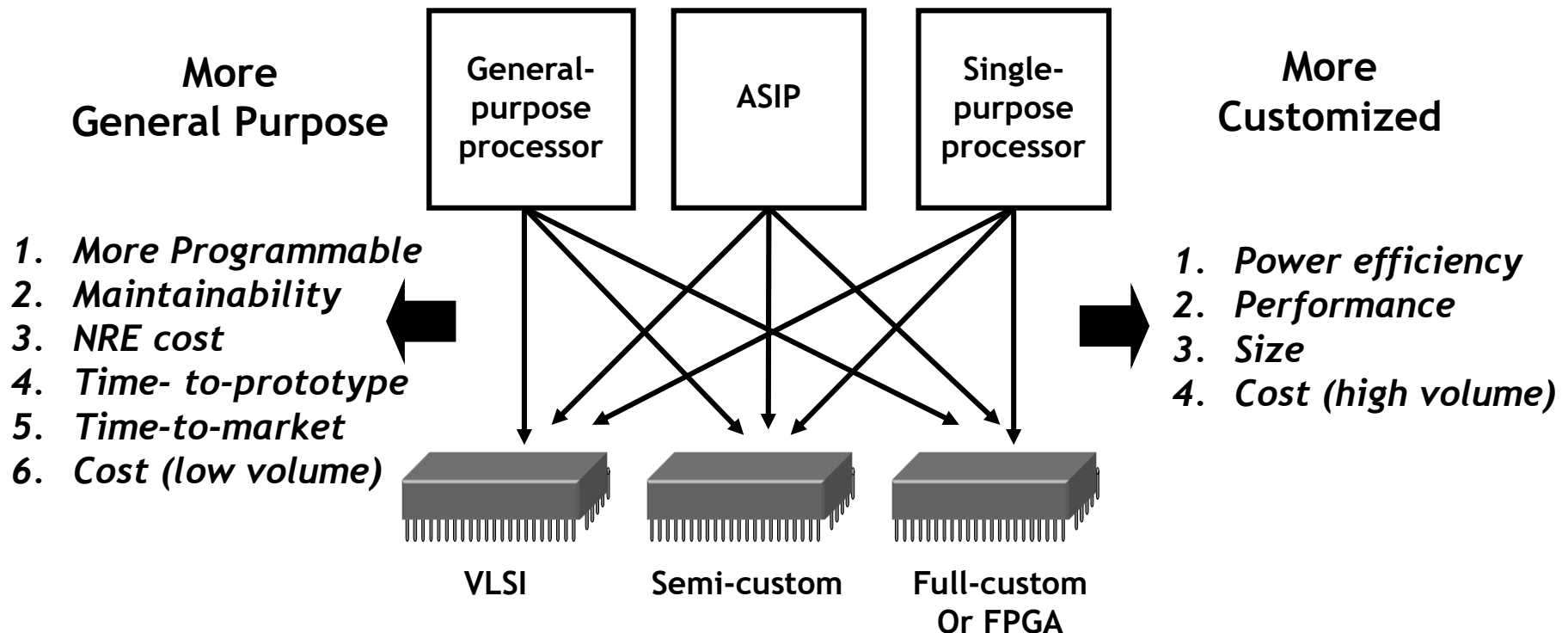


The choice of hardware versus software for a particular function is simply a tradeoff among various design metrics, like performance, power, size, NRE cost, and especially flexibility

there is no fundamental difference between what hardware or software can implement.

Independence of processor and technologies

- Basic tradeoff
 - General vs. custom
 - With respect to processor technology or IC technology
 - The two technologies are independent



Outline

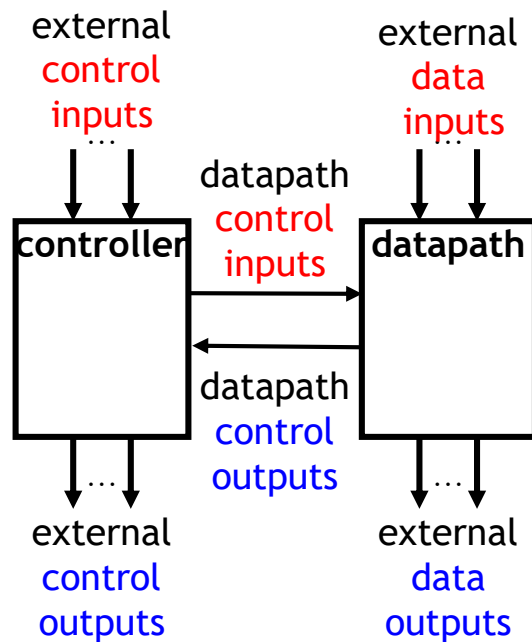
1. What is a processor?

2. How can we design the processor?

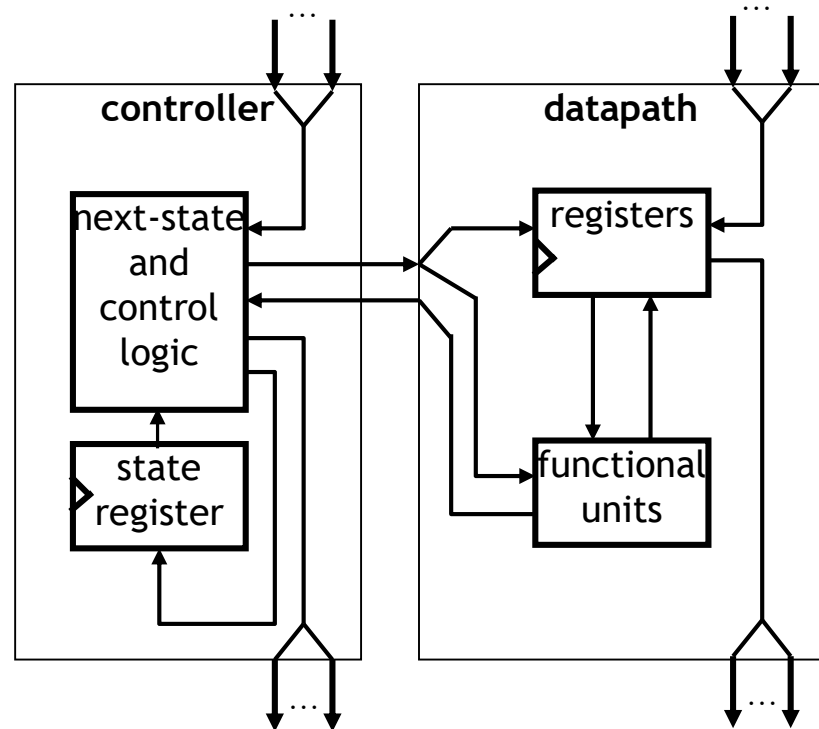
- **Single purpose processor Design**
- *Optimization*
- *Architecture-level Design*
- *System-level Design*

3. Brain Board

Custom single-purpose processor basic model



controller and datapath



a view inside the controller and datapath

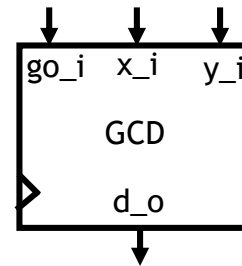
Example: greatest common divisor(GCD)

(最大公測度 or 最大公約數)

1. First create algorithm
2. Convert algorithm to “complex” state machine

- Known as **FSMD**: finite-state machine with datapath
- Can use **templates** to perform such conversion

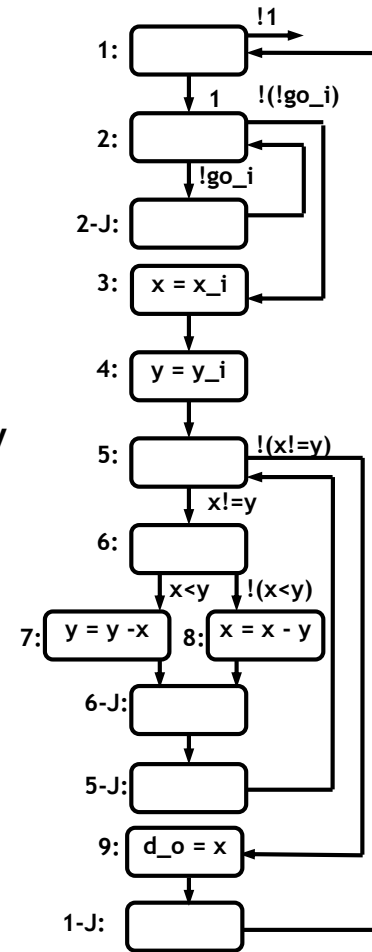
(a) black-box view



(b) desired functionality

```
0: int x, y;  
1: while (1) {  
2:   while (!go_i);  
3:   x = x_i;  
4:   y = y_i;  
5:   while (x != y) {  
6:     if (x < y)  
7:       y = y - x;  
8:       x = x - y;  
9:   }  
9:   d_o = x;  
}
```

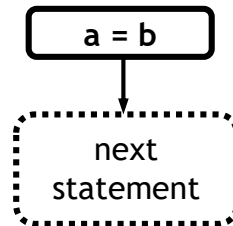
(c) state diagram



State diagram templates

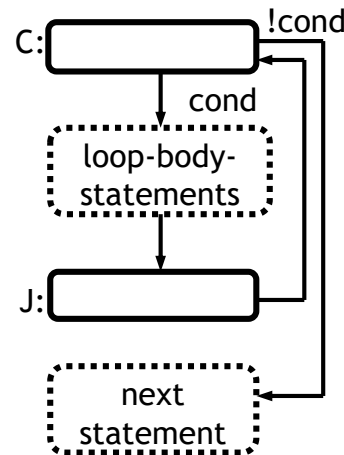
Assignment statement

a = b
next
statement



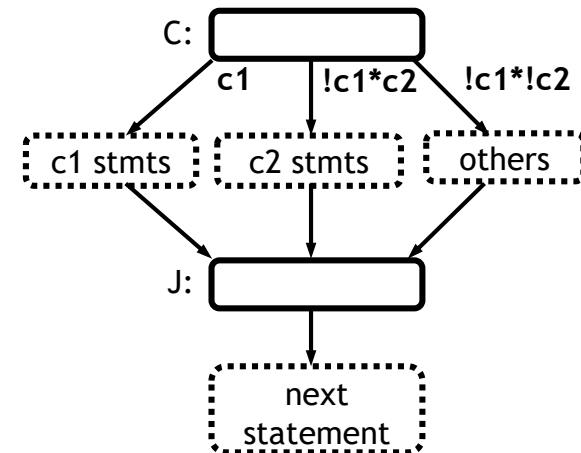
Loop statement

while (cond) {
loop-body-
statements
}
next statement



Branch statement

if (c1)
c1 stmts
else if c2
c2 stmts
else
other stmts
next statement

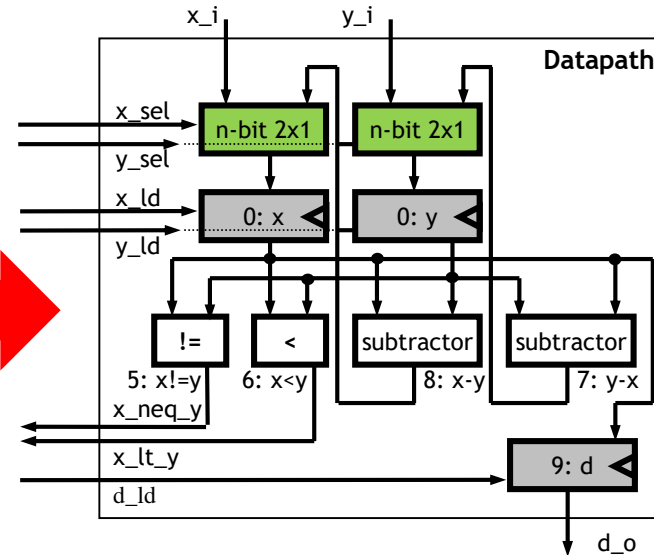
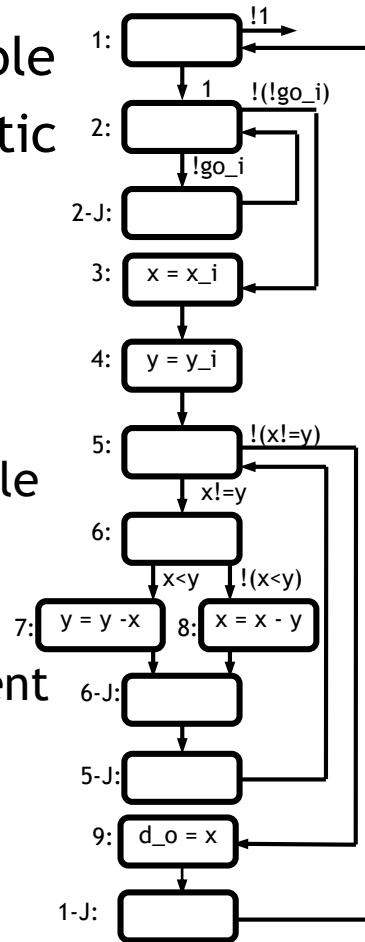


```

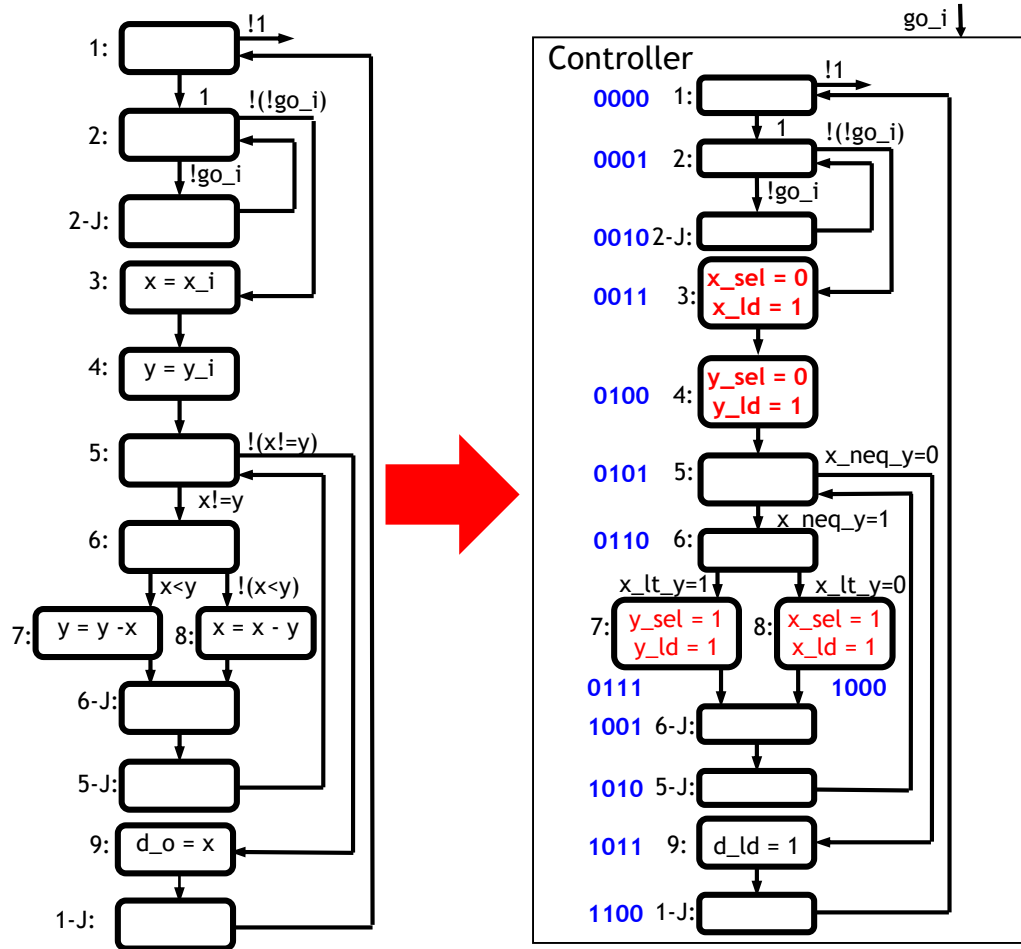
0: int x, y;
1: while (1) {
2:   while (!go_i);
3:   x = x_i;
4:   y = y_i;
5:   while (x != y) {
6:     if (x < y)
7:       y = y - x;
8:     else
9:       x = x - y;
   }
   d_o = x;
}
  
```


Creating the datapath

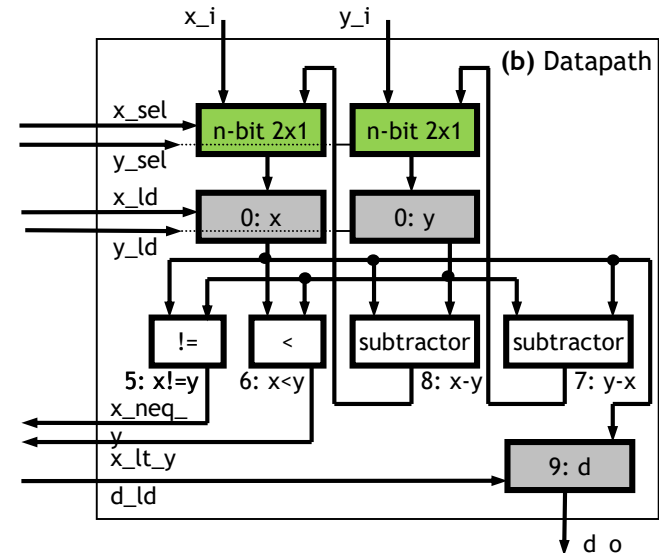
- a register \leftarrow declared variable
- a functional unit \leftarrow arithmetic operation
- Connect the ports, registers and functional units
 - Based on reads and writes
 - Use multiplexors for multiple sources
- Create unique identifier
 - for each datapath component control input and output



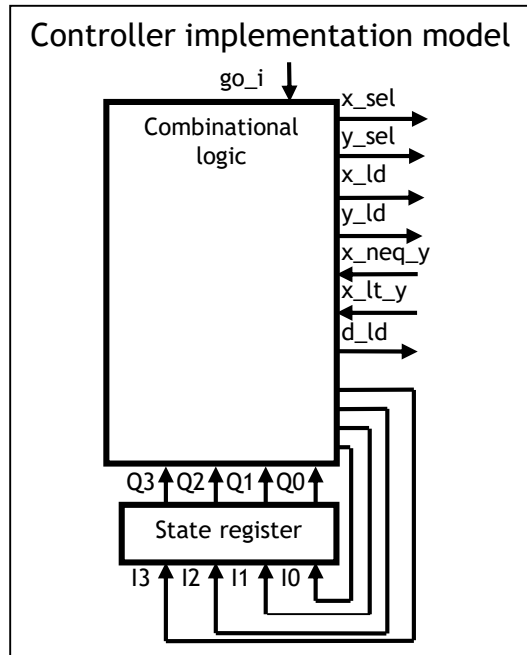
Creating the controller's FSM



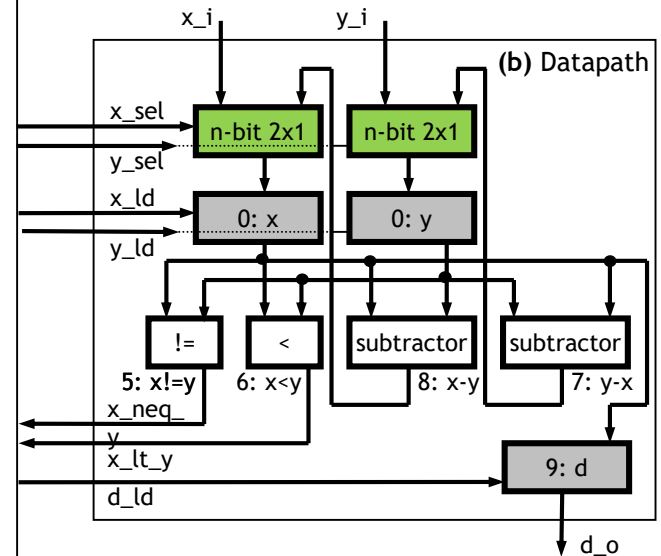
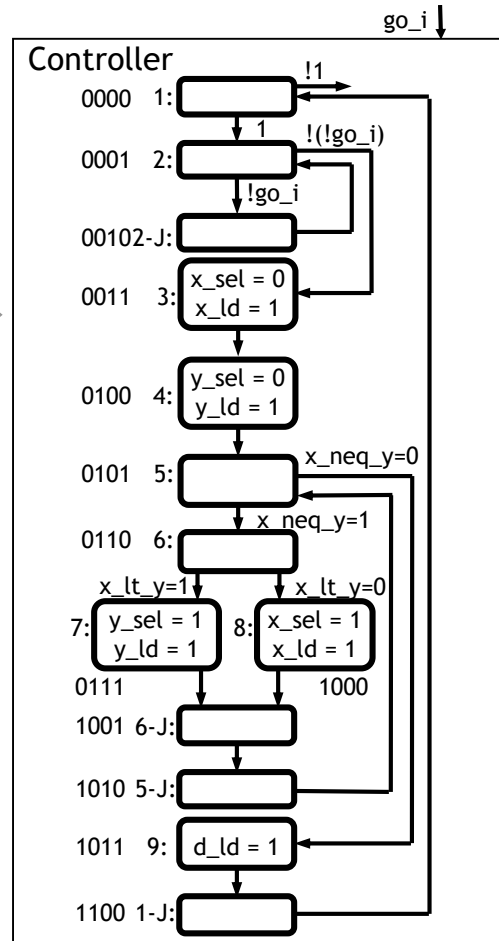
- Same structure as FSMD
- Replace complex actions/conditions with datapath configurations



Splitting into a controller and datapath



→

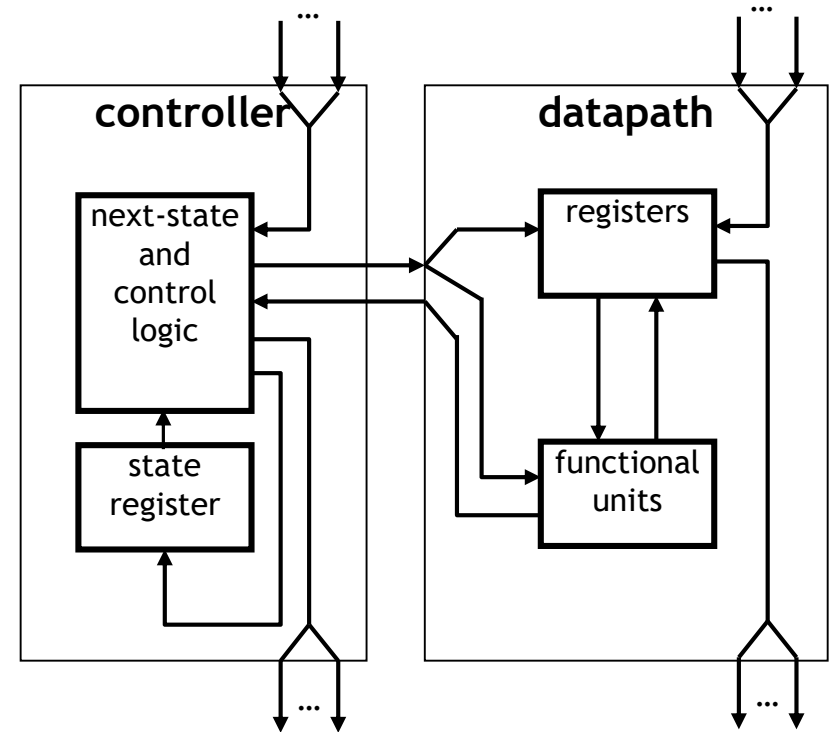


Controller state table for the GCD example

7 Inputs							9 Outputs								
Q3	Q2	Q1	Q0	x_neq_y	x_lt_y	go_i	I3	I2	I1	I0	x_sel	y_sel	x_ld	y_ld	d_ld
0	0	0	0	*	*	*	0	0	0	1	X	X	0	0	0
0	0	0	1	*	*	0	0	0	1	0	X	X	0	0	0
0	0	0	1	*	*	1	0	0	1	1	X	X	0	0	0
0	0	1	0	*	*	*	0	0	0	1	X	X	0	0	0
0	0	1	1	*	*	*	0	1	0	0	0	X	1	0	0
0	1	0	0	*	*	*	0	1	0	1	X	0	0	1	0
0	1	0	1	0	*	*	1	0	1	1	X	X	0	0	0
0	1	0	1	1	*	*	0	1	1	0	X	X	0	0	0
0	1	1	0	*	0	*	1	0	0	0	X	X	0	0	0
0	1	1	0	*	1	*	0	1	1	1	X	X	0	0	0
0	1	1	1	*	*	*	1	0	0	1	X	1	0	1	0
1	0	0	0	*	*	*	1	0	0	1	1	X	1	0	0
1	0	0	1	*	*	*	1	0	1	0	X	X	0	0	0
1	0	1	0	*	*	*	0	1	0	1	X	X	0	0	0
1	0	1	1	*	*	*	1	1	0	0	X	X	0	0	1
1	1	0	0	*	*	*	0	0	0	0	X	X	0	0	0
1	1	0	1	*	*	*	0	0	0	0	X	X	0	0	0
1	1	1	0	*	*	*	0	0	0	0	X	X	0	0	0
1	1	1	1	*	*	*	0	0	0	0	X	X	0	0	0

GCD custom single-purpose processor

- We finished the datapath
- We have a state table for the next state and control logic
 - All that's left is combinational logic design
- This is *not* an optimized design, but we see the basic steps
- Usually, we design with RTL and use CAD SW to convert RTL into controller and datapath



a view inside the controller and datapath

Outline

1. What is a processor?

2. How can we design the processor?

- *Single purpose processor Design*
- **Optimization**
 - *For better logic design*
- *Architecture-level Design*
- *System-level Design*

3. Brain Board

Optimizing single-purpose processors

- Optimization is the task of making design metric values the best possible
- Optimization opportunities
 1. Original program
 2. FSMD
 3. Datapath
 4. FSM

Optimizing the original program

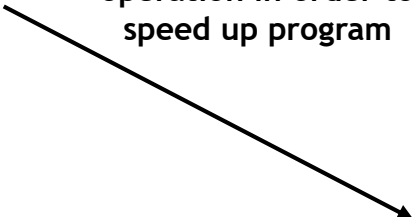
- Analyze program attributes and look for areas of possible improvement
 - number of computations
 - size of variables
 - time and space complexity
 - operations used
 - multiplication and division are very expensive

Optimizing 1) the original program

original program

```
0: int x, y;
1: while (1) {
2:   while (!go_i);
3:   x = x_i;
4:   y = y_i;
5:   while (x != y) {
6:     if (x < y)
7:       y = y - x;
8:     else
9:       x = x - y;
10:  }
11:  d_o = x;
12: }
```

replace the subtraction
operation(s) with modulo
operation in order to
speed up program



optimized program

```
0: int x, y, r;
1: while (1) {
2:   while (!go_i);
3:   // x must be the larger
4:   // number
5:   if (x_i >= y_i) {
6:     x=x_i;
7:     y=y_i;
8:   }
9:   else {
10:    x=y_i;
11:    y=x_i;
12:  }
13:  while (y != 0) {
14:    r = x % y;
15:    x = y;
16:    y = r;
17:  }
18:  d_o = x;
19: }
```

GCD(42, 8) - 9 iterations to complete the loop

x and y values evaluated as follows : (42, 8), (43, 8),
(26,8), (18,8), (10, 8), (2,8), (2,6), (2,4), (2,2).

GCD(42, 8) - 3 iterations to complete the loop

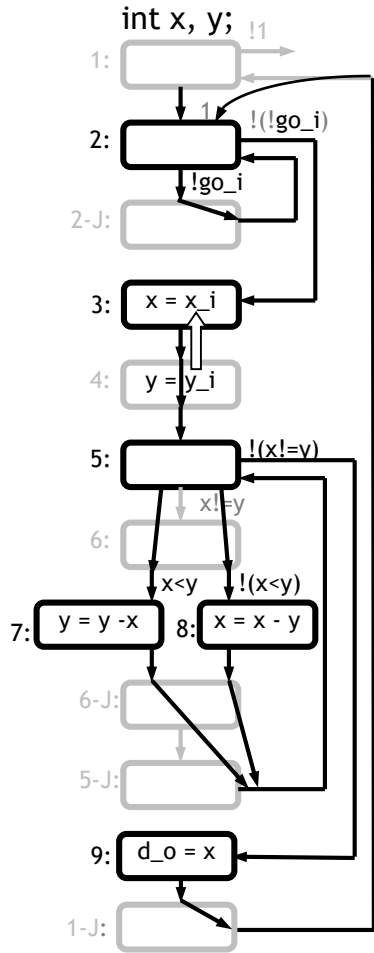
x and y values evaluated as follows: (42, 8),
(8,2), (2,0)

Optimizing 2) the FSMD

- Areas of possible improvements
 - merge states
 - states with constants on transitions can be eliminated, transition taken is already known
 - states with independent operations can be merged
 - separate states
 - states which require complex operations ($a*b*c*d$) can be broken into smaller states to reduce hardware size
 - scheduling

Optimizing 2) the FSMD (cont.)

Original FSMD



eliminate state 1 - transitions have constant values

merge state 2 and state 2J - no loop operation in between them

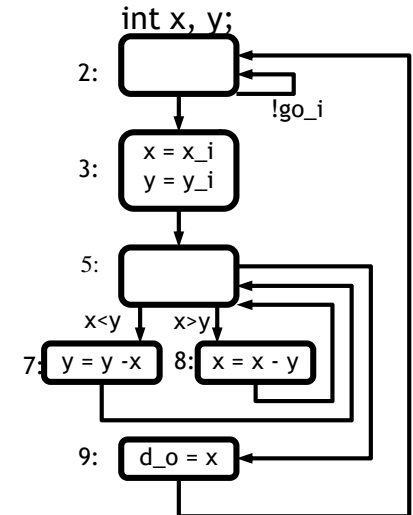
merge state 3 and state 4 - assignment operations are independent of one another

merge state 5 and state 6 - transitions from state 6 can be done in state 5

eliminate state 5J and 6J - transitions from each state can be done from state 7 and state 8, respectively

eliminate state 1-J - transition from state 1-J can be done directly from state 9

Optimized FSMD



Optimizing 3) the datapath

- Sharing of functional units
 - one-to-one mapping, as done previously, is not necessary
 - if same operation occurs in different states, they can share a single functional unit
- Multi-functional units
 - ALUs support a variety of operations, it can be shared among operations occurring in different states

Optimizing 4) the FSM

- State encoding
 - task of assigning a unique bit pattern to each state in an FSM
 - size of state register and combinational logic vary
 - can be treated as an ordering problem
- State minimization
 - task of merging equivalent states into a single state
 - state equivalent if for all possible input combinations which the two states generate are the same outputs and transitions to the next same state

SW in General Purpose Processor

- General-Purpose Processor
 - Processor designed for a variety of computation tasks
 - Low unit cost, in part because manufacturer spreads NRE over large numbers of units
 - Intel, AMD, ARM-based CPUs
 - Carefully designed since higher NRE is acceptable
 - Can yield good performance, size and power
 - Low NRE cost, short time-to-market/prototype, high flexibility
 - User just writes software; no processor design
 - a.k.a. “microprocessor” - “micro” used when they were implemented on one or a few chips rather than entire rooms

Outline

1. What is a processor?

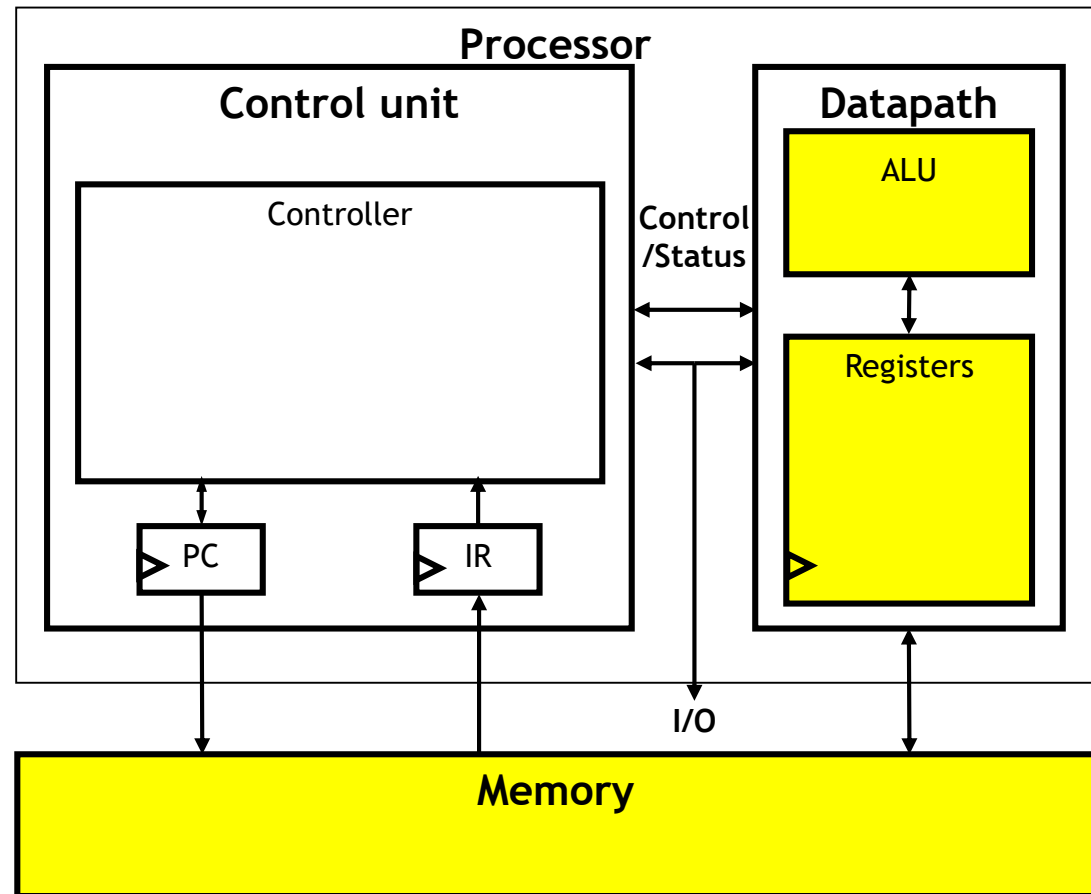
2. How can we design the processor?

- *Single purpose processor Design*
- *Optimization*
- **Architecture-level Design**
 - *How to link Processor for final “target” operations?*
- *System-level Design*

3. Brain Board

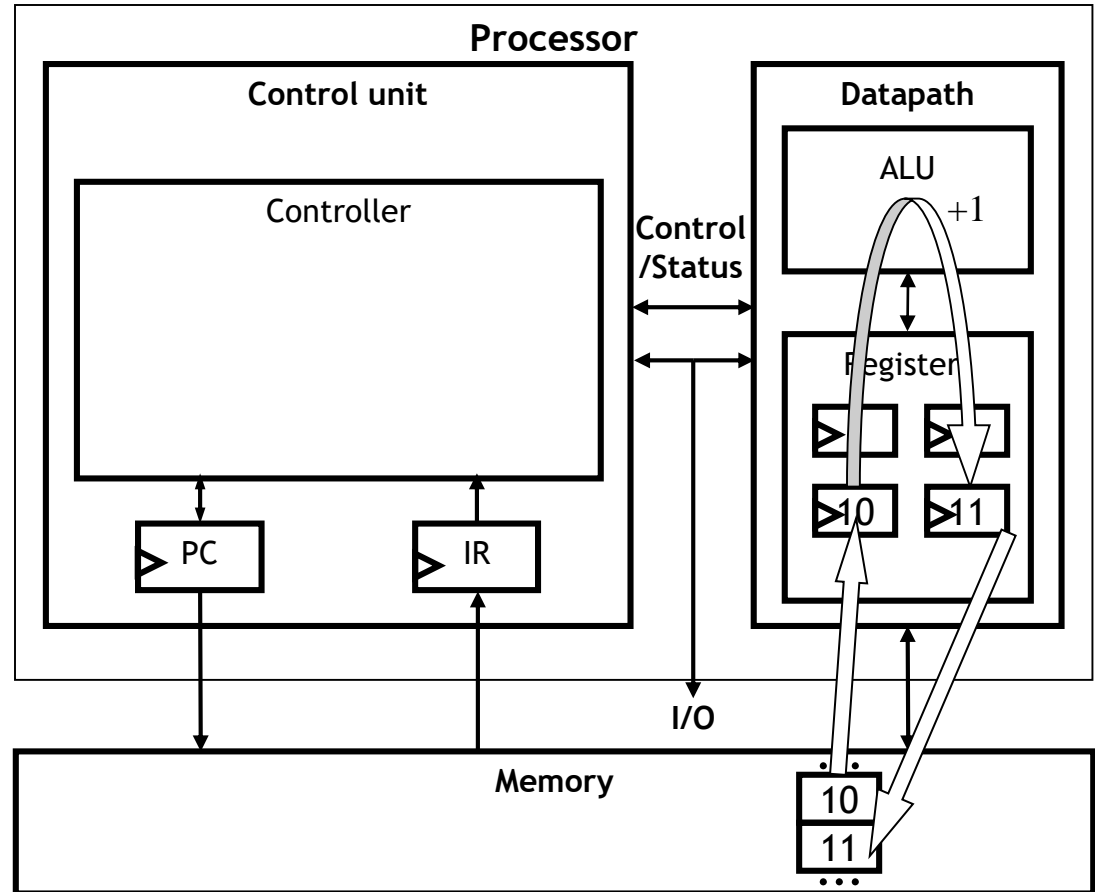
Basic Architecture of General Processor

- Control unit and Datapath
 - Note similarity to single-purpose processor
- Key differences
 - Datapath is general
 - Control unit doesn't store the algorithm - the algorithm is "programmed and stored" into the memory



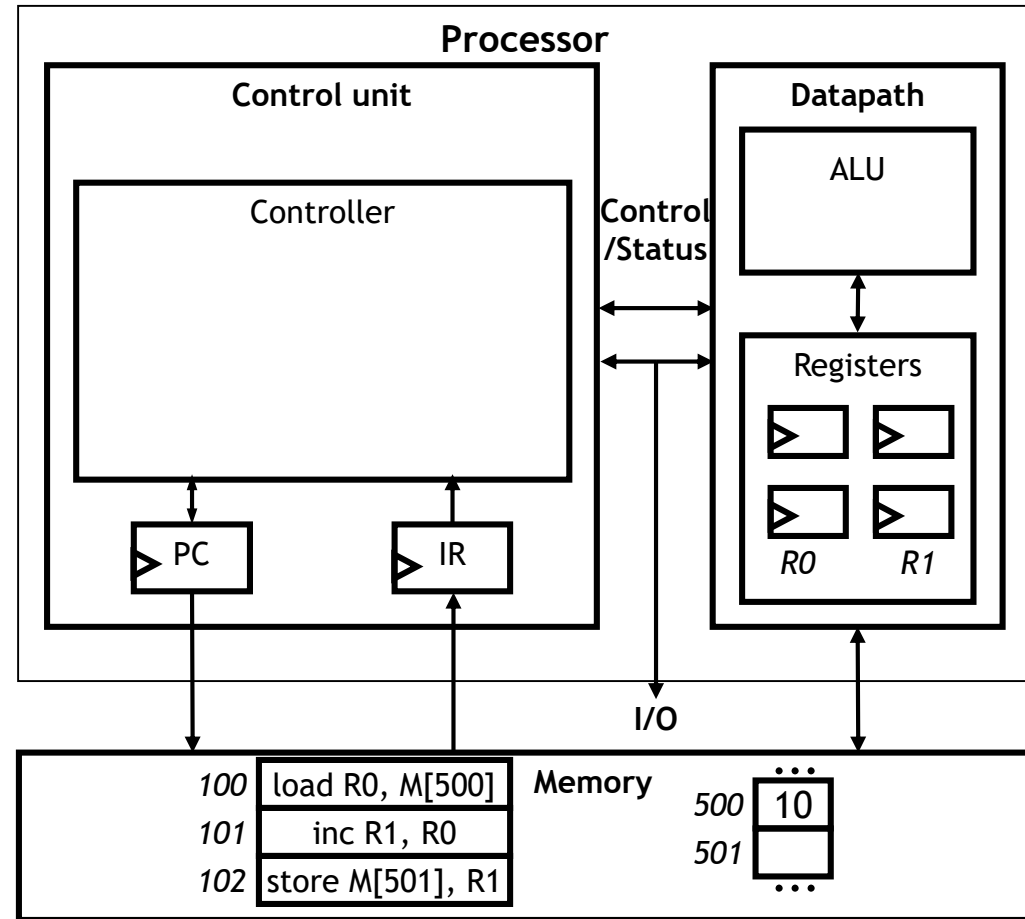
Datapath Operations

- Load
 - Read memory location into register
- ALU operation
 - Input certain registers through ALU, store back in register
- Store
 - Write register to memory location



Control Unit

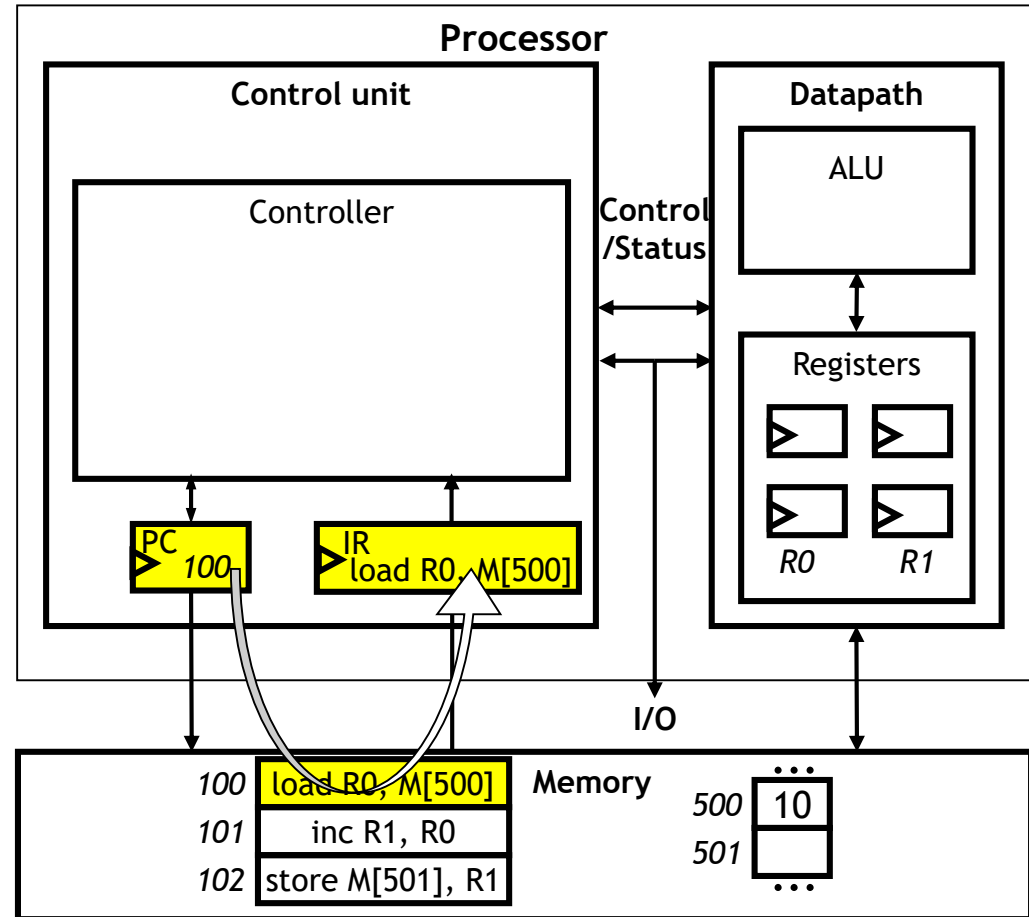
- Control unit: configures the datapath operations
 - Sequence of desired operations (“instructions”) stored in memory
 - “program”
- Instruction cycle - broken into several sub-operations, each one clock cycle, e.g.:
 - 1. I Fetch:** Get next instruction into IR
 - 2. I Decode:** Determine what the instruction means
 - 3. Fetch operands from Memory:** Move data from memory to datapath register
 - 4. Execute:** Move data through the ALU
 - 5. Write back results:** Write data from register to memory



Control Unit Sub-Operations

- **I Fetch (IF)**

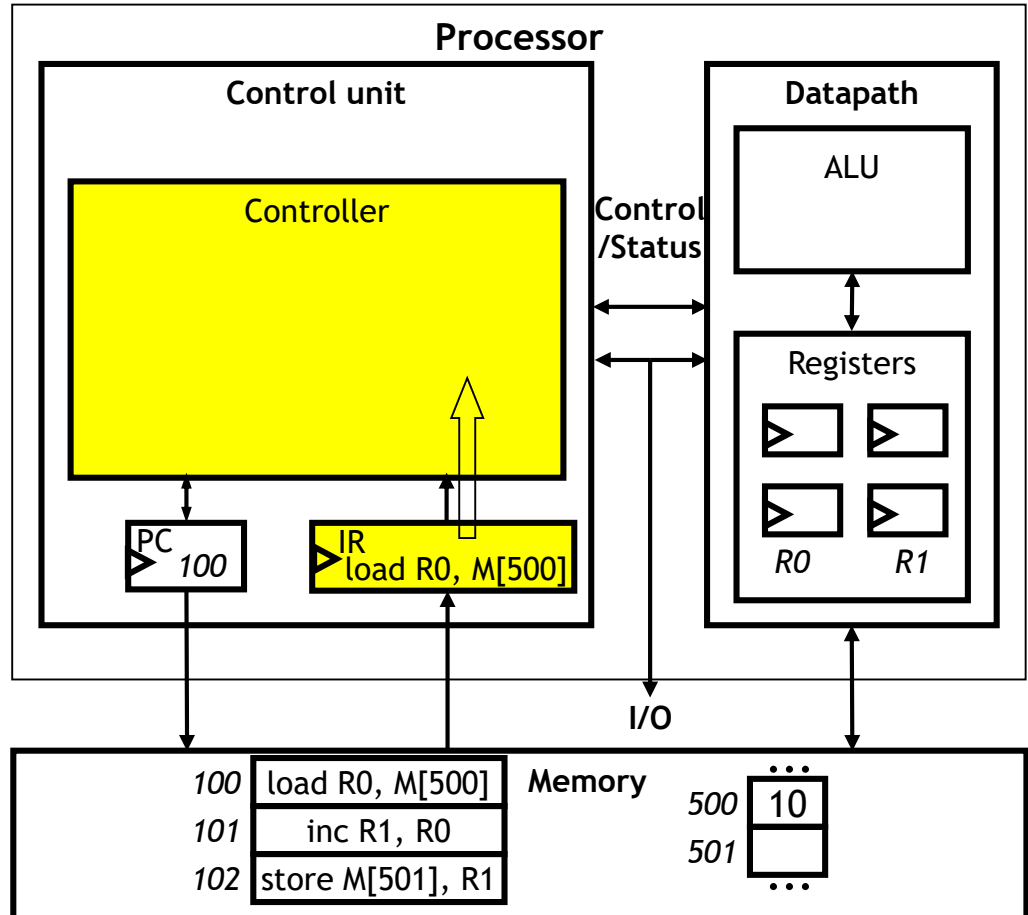
- Get next instruction into IR
- PC: program counter, always points to next instruction
- IR: holds the fetched instruction



Control Unit Sub-Operations

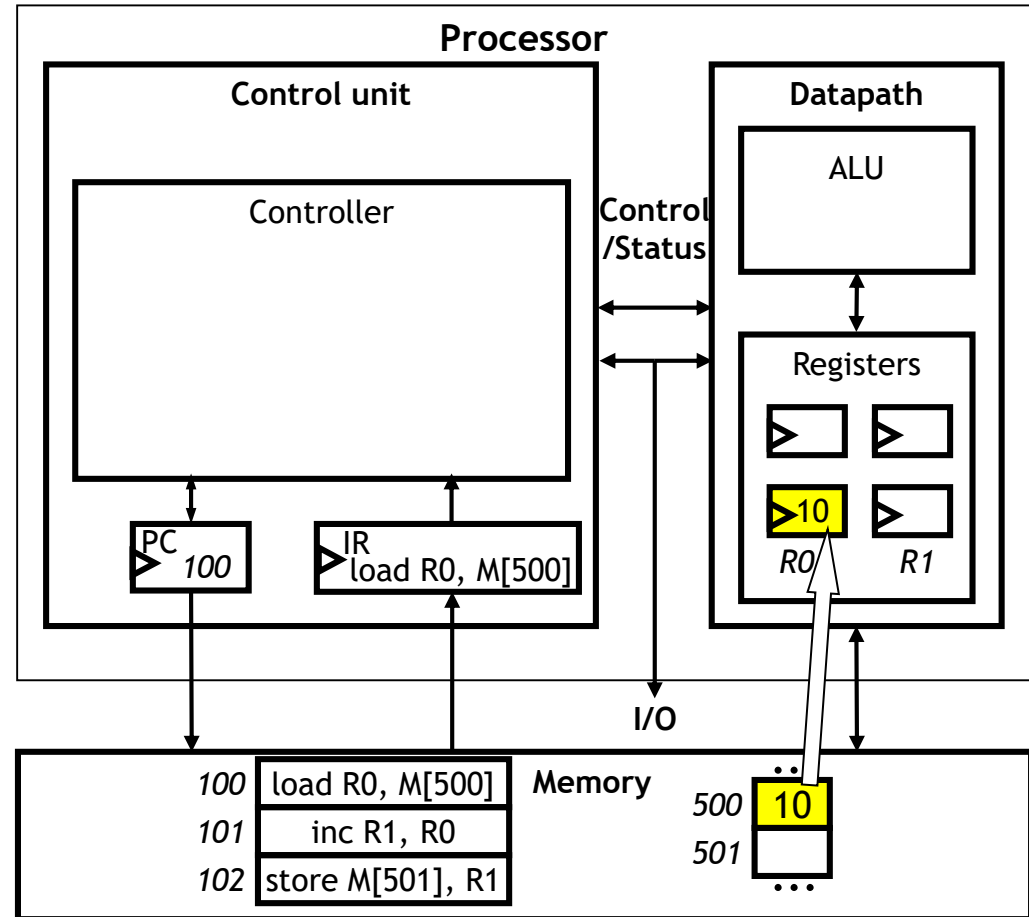
- **I Decode (ID)**

- Determine what the instruction means



Control Unit Sub-Operations

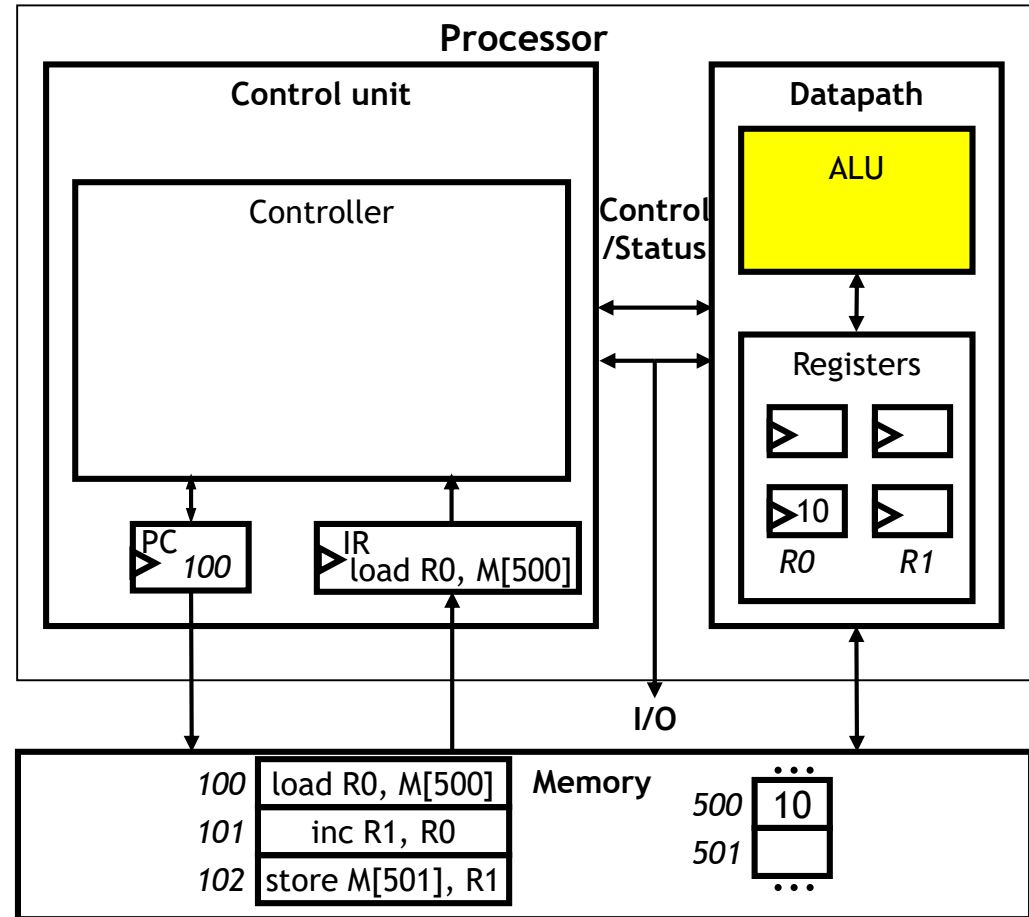
- Fetch operands from Memory (Mem)
 - Move data from memory to datapath register



Control Unit Sub-Operations

- **Execute (EX)**

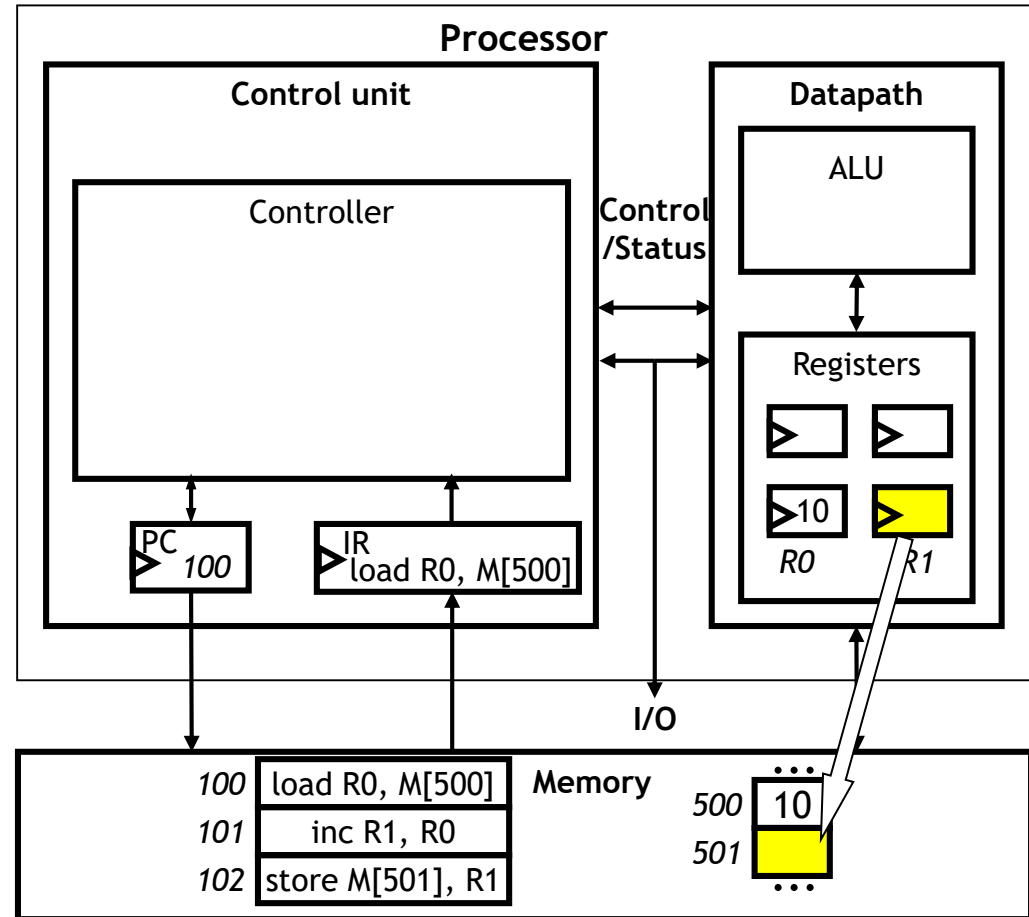
- Move data through the ALU
- This particular instruction does nothing during this sub-operation



Control Unit Sub-Operations

- Write back results (WB)

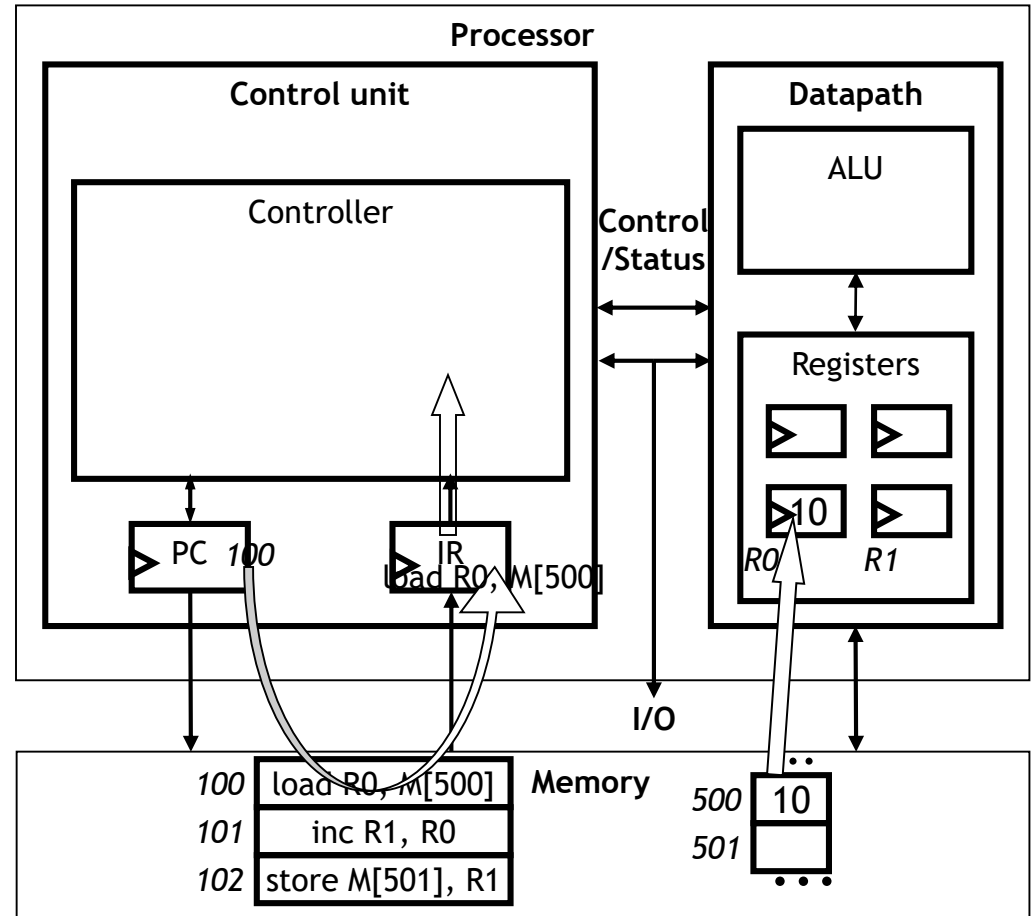
- Write data from register to memory
- This particular instruction does nothing during this sub-operation



Instruction Cycles

PC=100

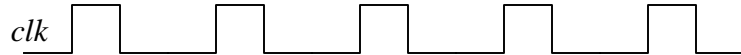
IF ID Mem EX WB



Instruction Cycles

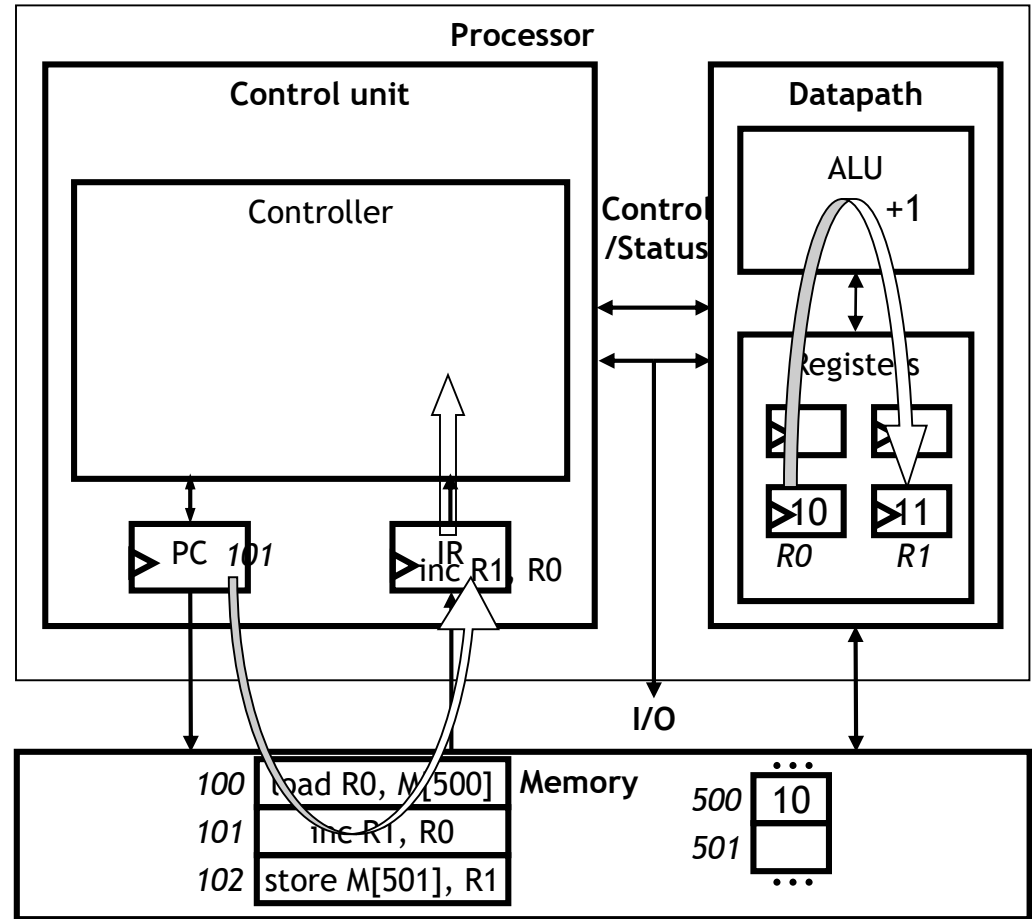
PC=100

IF ID Mem EX WB



PC=101

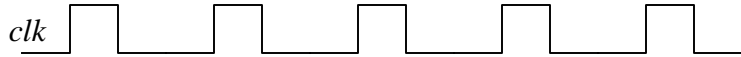
IF ID Mem EX WB



Instruction Cycles

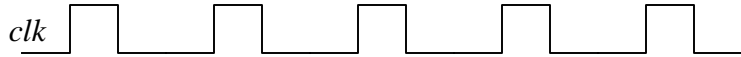
PC=100

IF ID Mem EX WB



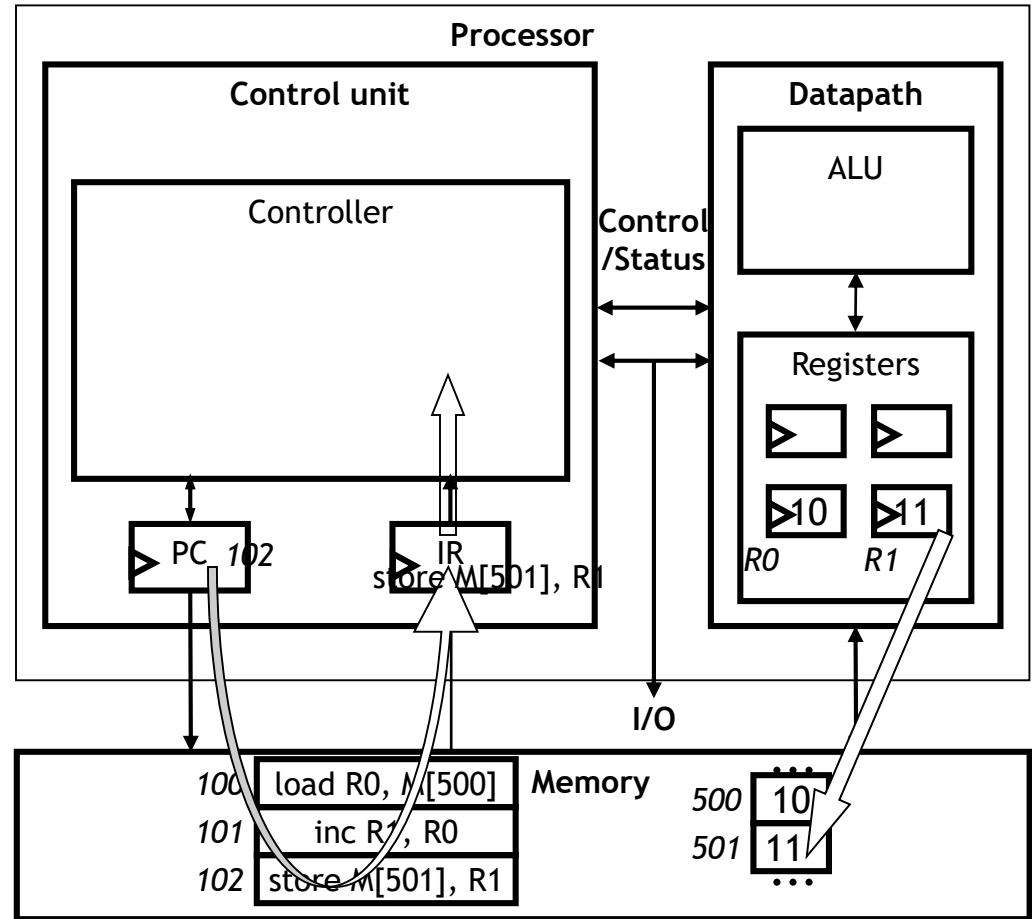
PC=101

IF ID Mem EX WB



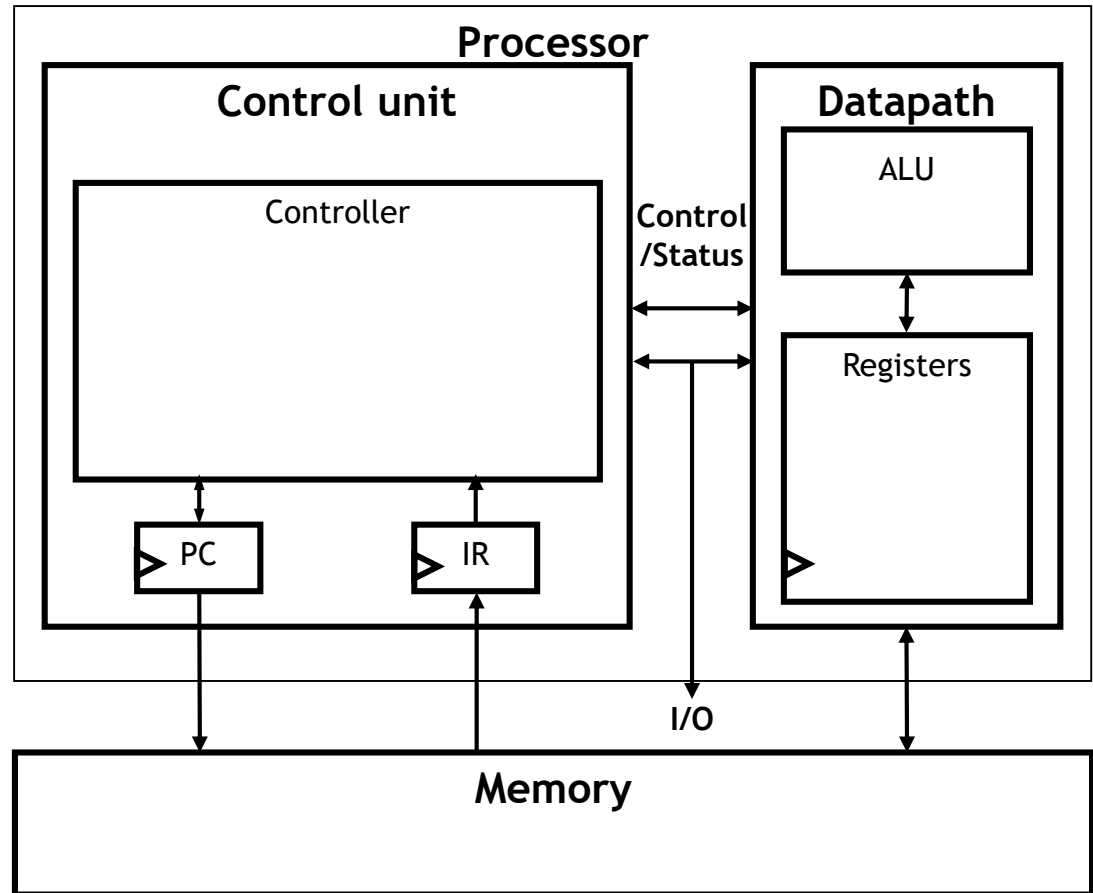
PC=102

IF ID Mem EX WB



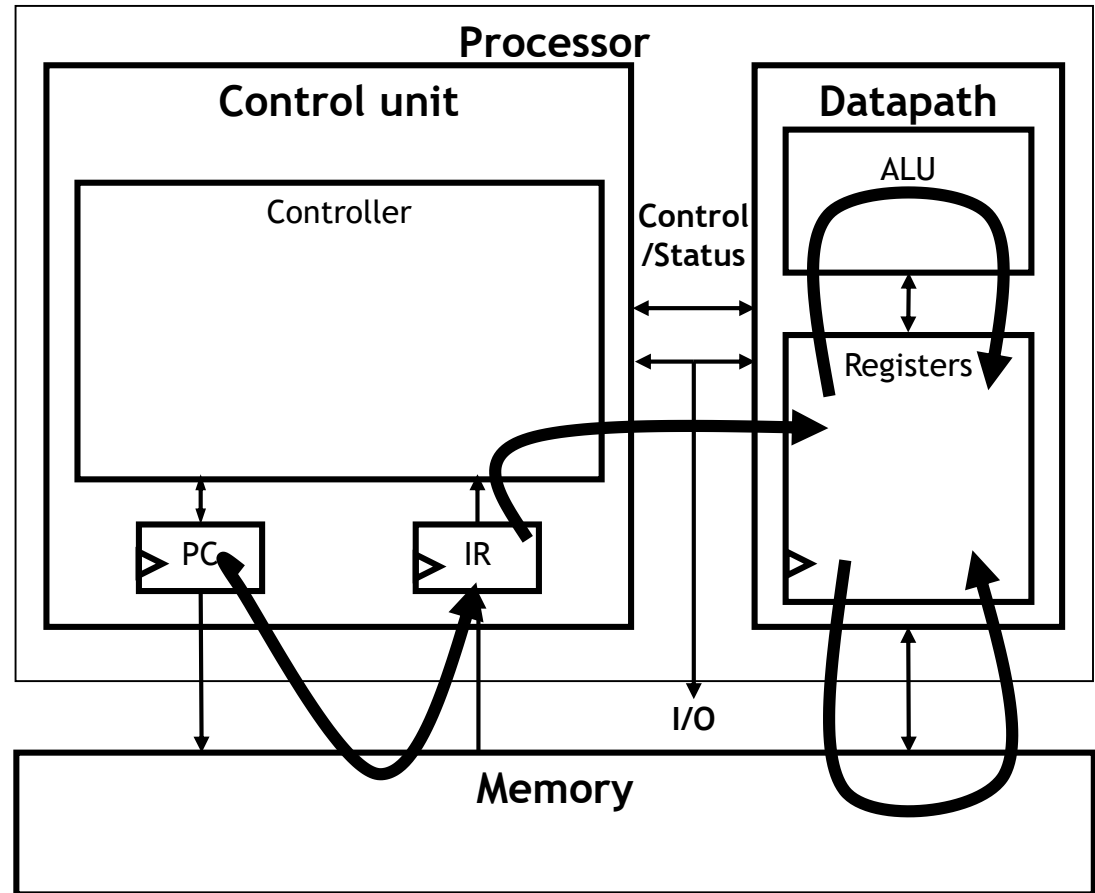
Architectural Considerations

- *N-bit* processor
 - N-bit ALU, registers, buses, memory data interface
 - Embedded: 8-bit, 16-bit, 32-bit common
 - Desktop/servers: 32-bit, even 64
- PC size determines address space



Architectural Considerations

- Clock frequency
 - Inverse of clock period of time
 - Must be longer than longest register to register delay in entire processor
 - Memory access is often the longest



Pipelining: Increasing Instruction *Throughput*



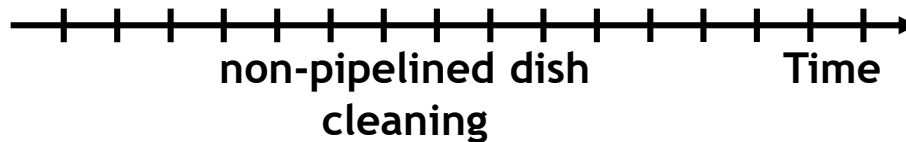
Wash



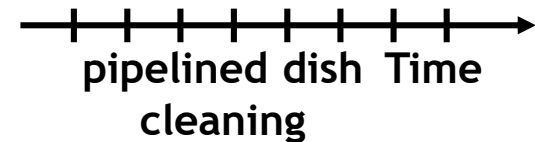
Dry



Non-pipelined



Pipelined



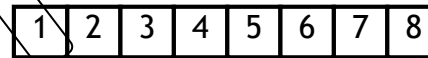
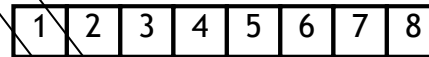
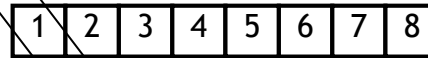
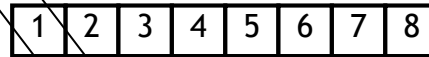
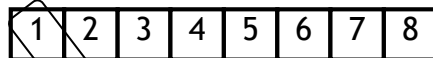
Fetch-instr.

Decode

Fetch ops.

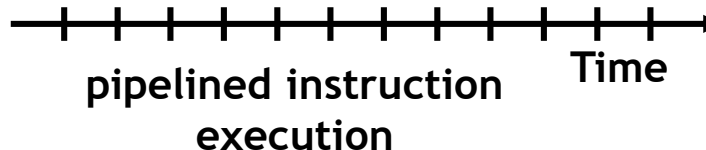
Execute

Store res.



Instruction 1

Pipelined



Programmer's View

- Programmer doesn't need detailed understanding of architecture
 - Instead, needs to know what instructions can be executed
- Two levels of instructions:
 - Assembly level
 - Structured languages (C, C++, Java, etc.)
- Most development today done using structured languages
 - But, some assembly level programming may still be necessary
 - Drivers: portion of program that communicates with and/or controls (drives) another device
 - Often have detailed timing considerations, extensive bit manipulation
 - Assembly level may be best for these

Assembly-Level Instructions

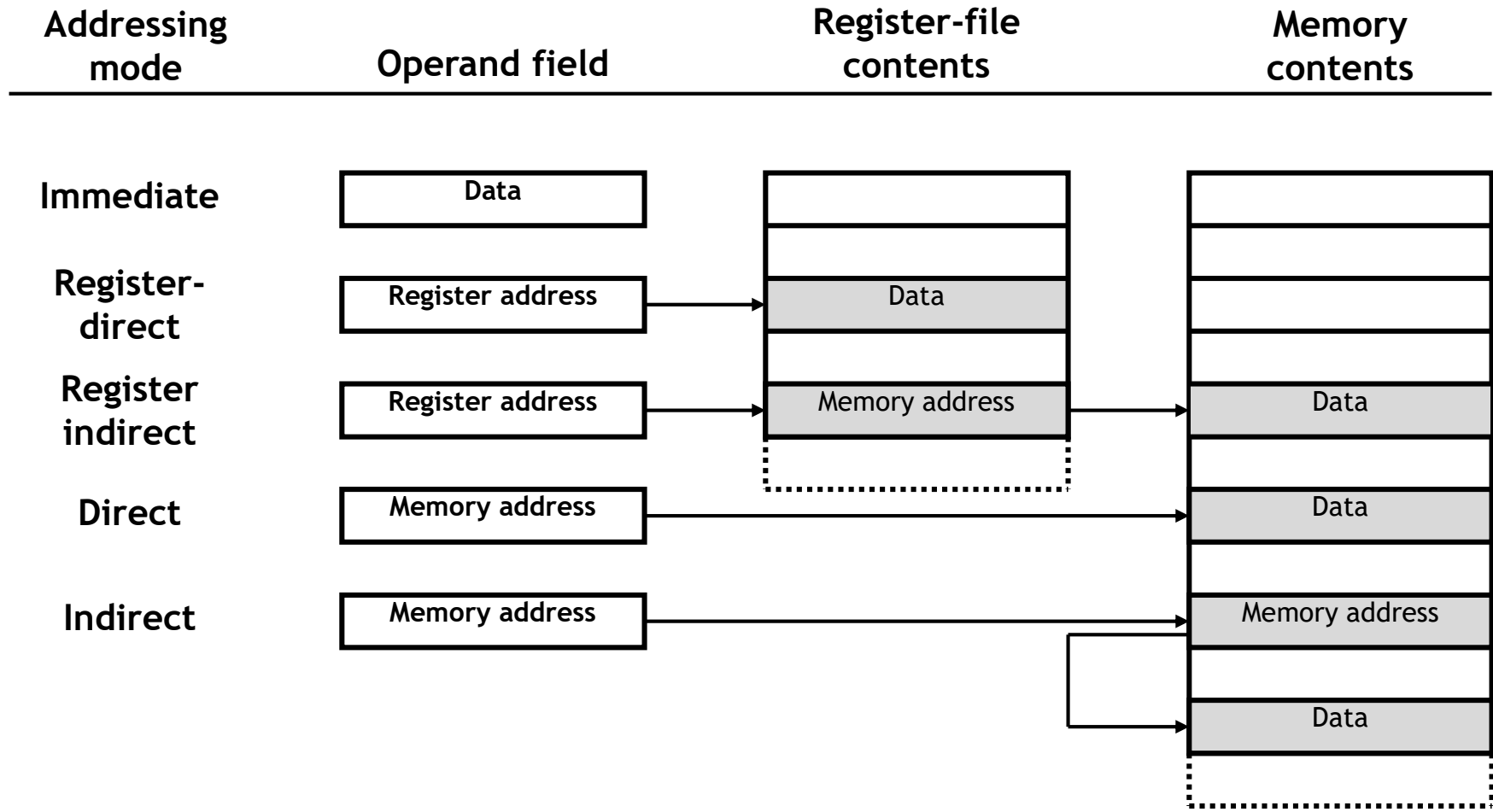
Instruction 1	opcode	operand1	operand2
Instruction 2	opcode	operand1	operand2
Instruction 3	opcode	operand1	operand2
Instruction 4	opcode	operand1	operand2
...			

- Instruction Set
 - Defines the legal set of instructions for that processor
 1. **Data transfer**: memory/register, register/register, I/O, etc.
 2. **Arithmetic/logical**: move register through ALU and back
 3. **Branches**: determine next PC value when not just PC+1

A Simple (Trivial) Instruction Set

Assembly instruct.	First byte		Second byte	Operation
MOV Rn, direct	0000	Rn	direct	$Rn = M(\text{direct})$
MOV direct, Rn	0001	Rn	direct	$M(\text{direct}) = Rn$
MOV @Rn, Rm	0010	Rn	Rm	$M(Rn) = Rm$
MOV Rn, #immed.	0011	Rn	immediate	$Rn = \text{immediate}$
ADD Rn, Rm	0100	Rn	Rm	$Rn = Rn + Rm$
SUB Rn, Rm	0101	Rn	Rm	$Rn = Rn - Rm$
JZ Rn, relative	0110	Rn	relative	$PC = PC + \text{relative}$ (only if Rn is 0)
	opcode		operands	

Addressing Modes



Programmer Considerations

- Program and data memory space
 - Embedded processors often very limited
 - e.g., 64 Kbytes program, 256 bytes of RAM (expandable)
- Registers: How many are there?
 - Only a direct concern for assembly-level programmers
- I/O
 - How communicate with external signals?
- Interrupts

Outline

1. What is a processor?

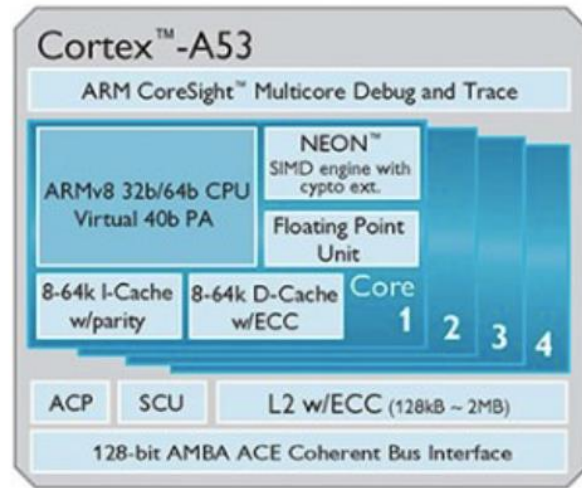
2. How can we design the processor?

- *Single purpose processor Design*
- *Optimization*
- **Architecture-level Design**
 - **ARM Architecture**
- *System-level Design*

3. Brain Board

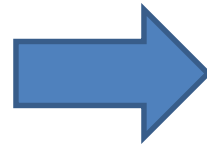
ARM designs the ‘processor’, not a chip

- 1) High power efficiency, 2) small code size,
- 3) Low power consumption, 4) small silicon area



Core design

- **ARM: Advanced RISC Machine**
- ARM licenses core designs to other companies
- OEMs can customize chips
- ARM provides compilers, development tools, and debugging tools for chip development and product development



Chip design & fabrication



Smartphone

ARM processors for embedded system

Cortex - A

Highest performance

Optimised for rich operating systems

The Cortex-A section features a green header and border. It includes icons for a steering wheel with a central display, a tablet showing a map, a black wireless router with three antennas, and a speedometer with the number 72.

Cortex - R

Fast response


Optimised for high performance, hard real-time applications

The Cortex-R section features a blue header and border. It includes icons for a 3.5-inch hard drive, a robotic arm with a gripper, and a car chassis with four wheels.

Cortex - M

Smallest/lowest power

Optimised for discrete processing and microcontrollers

The Cortex-M section features a purple header and border. It includes icons for a smartwatch, a headset with a microphone, and a lightbulb with green signal waves emanating from it.

Different design for diverse needs

High throughput, low latency, low power, ...

ARM (RISC) vs. x86 (CISC)

	ARM	x86
Architecture	RISC	CISC
Target Product	Low-power SoC & IP	High-end
Date Announced	1985 (ARM1)	1978 (80386)
# of Registers / Width	37 registers / 32bits	8 registers / <ul style="list-style-type: none">- IA-16 : 16bits- IA-32 : 32bits- AMD64 : 64bits
Instruction Length	Fixed length instructions (4 bytes)	Variable-length instructions (1 - 6 bytes)
Instruction Set Arch. (ISA)	Load-store architecture	Extended accumulator architecture
Example	MPC601, SPARC ARMv7-A (Cortex-A9) ARMv8-A (Cyclone, Mongoose)	MC68040 Intel 8086, 8088 (IA-16) Intel 80386 (IA-32) Pentium 4

Key differences in processors

RISC Philosophy

- Regularity & simplicity
- Leaner means faster
- Optimize the common case



ex) ARM

- Energy efficiency
- Embedded Systems
- Phones/Tablets

CISC Philosophy

- Compilers can be smart
- More Transistors
- Already PC market winner
- Code size counts so, Micro-code!



x86

- High Performance
- Desktops/Servers

ARM (Fixed-length Instruction) vs. x86 (Variable-length Instruction)

- Fixed-length instruction (32bits)
It makes simple hardware implementation & efficient execution
➔ Instead, Needs complex compiler

Uniformity ↑

➔ Multiple instruction fetch in parallel (superscalar)

- Variable-length instruction
ISA does as much as possible using hardware circuitry
Single instruction is decoded into many microcodes

ARM (Load-store Arch.) vs. x86 (Register-memory Arch.)

- Load-store Architecture

Only Load/Store can access to memory. Other operations access to register-register

Ex) `lw $t0, 12($gp)`

`add $s0, $s0, $t0 # s0 = s0 + Mem[12+gp]`

➔Simpler HW (Easy to pipeline, parallel computing)

- Register-memory Architecture

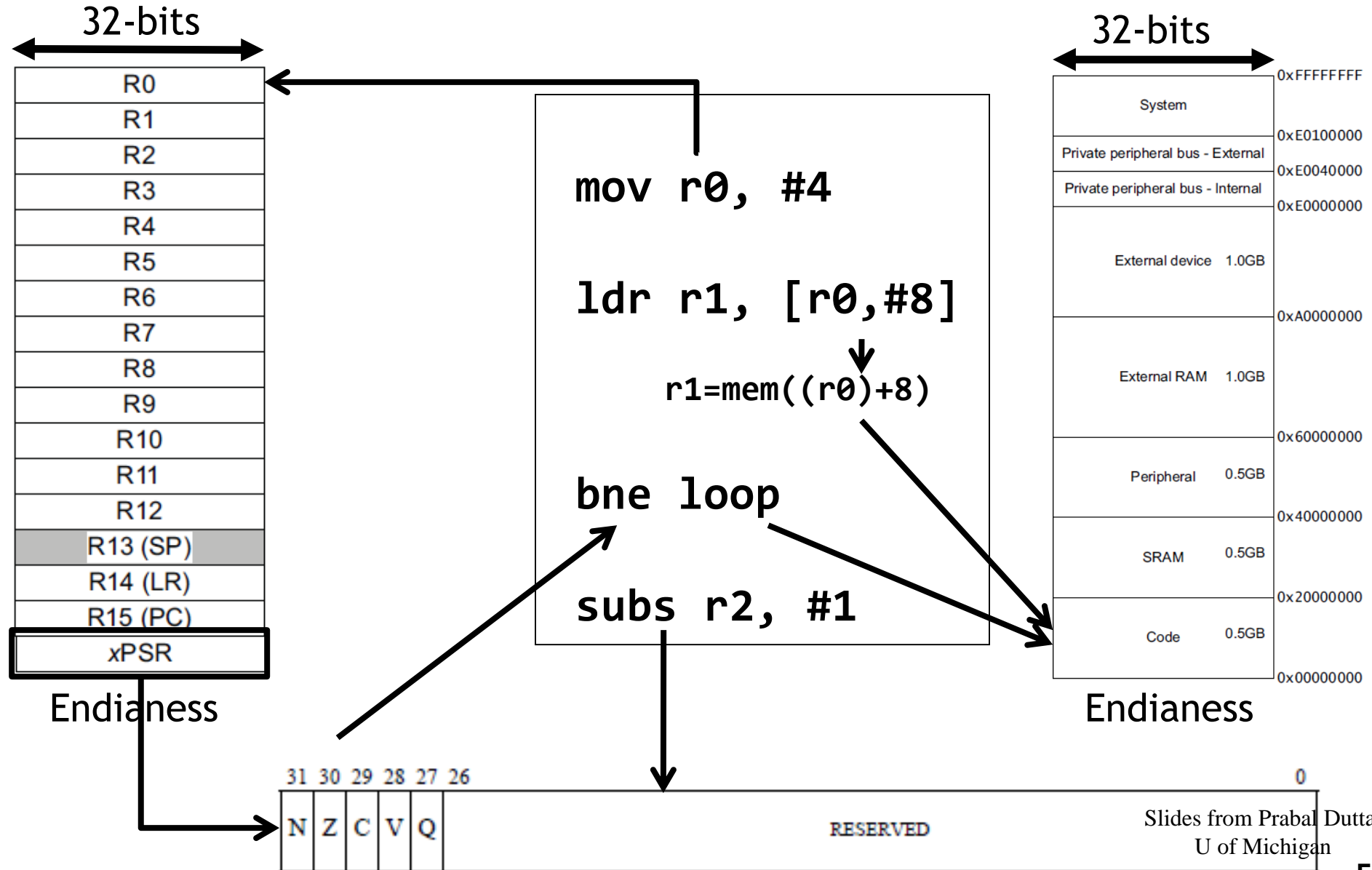
All operations can have an operand in memory

Ex) `add 12(%gp), $s0 # s0 = s0 + Mem[12+gp]`

➔ Few instructions ➔ Smaller codes

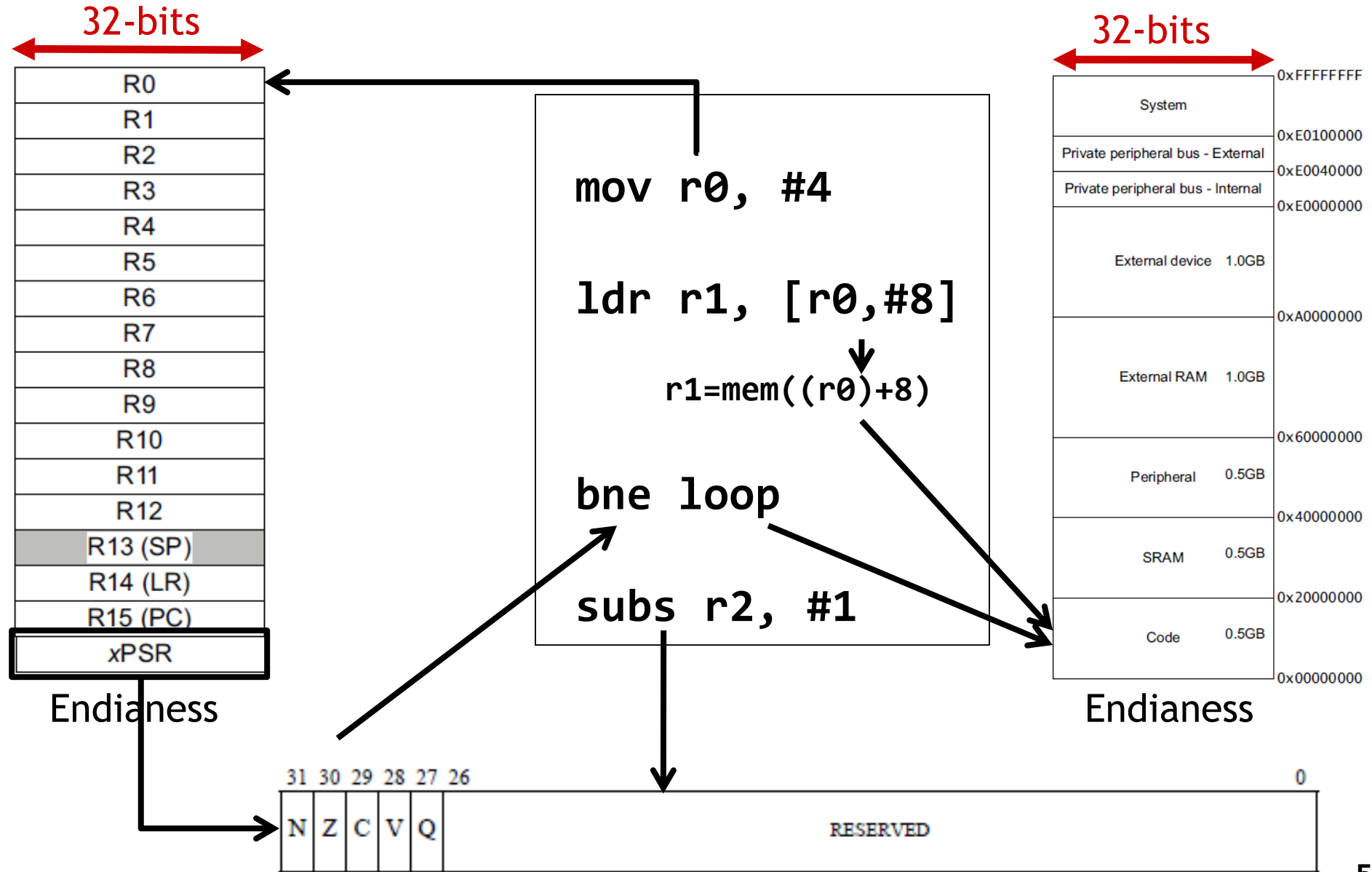
Introduction to ARM ISA

(word size, registers, memory, endianness, conditions, instructions, addressing modes)



Introduction to ARM ISA

(**word size**, registers, memory, endianness, conditions, instructions, addressing modes)



Word Size

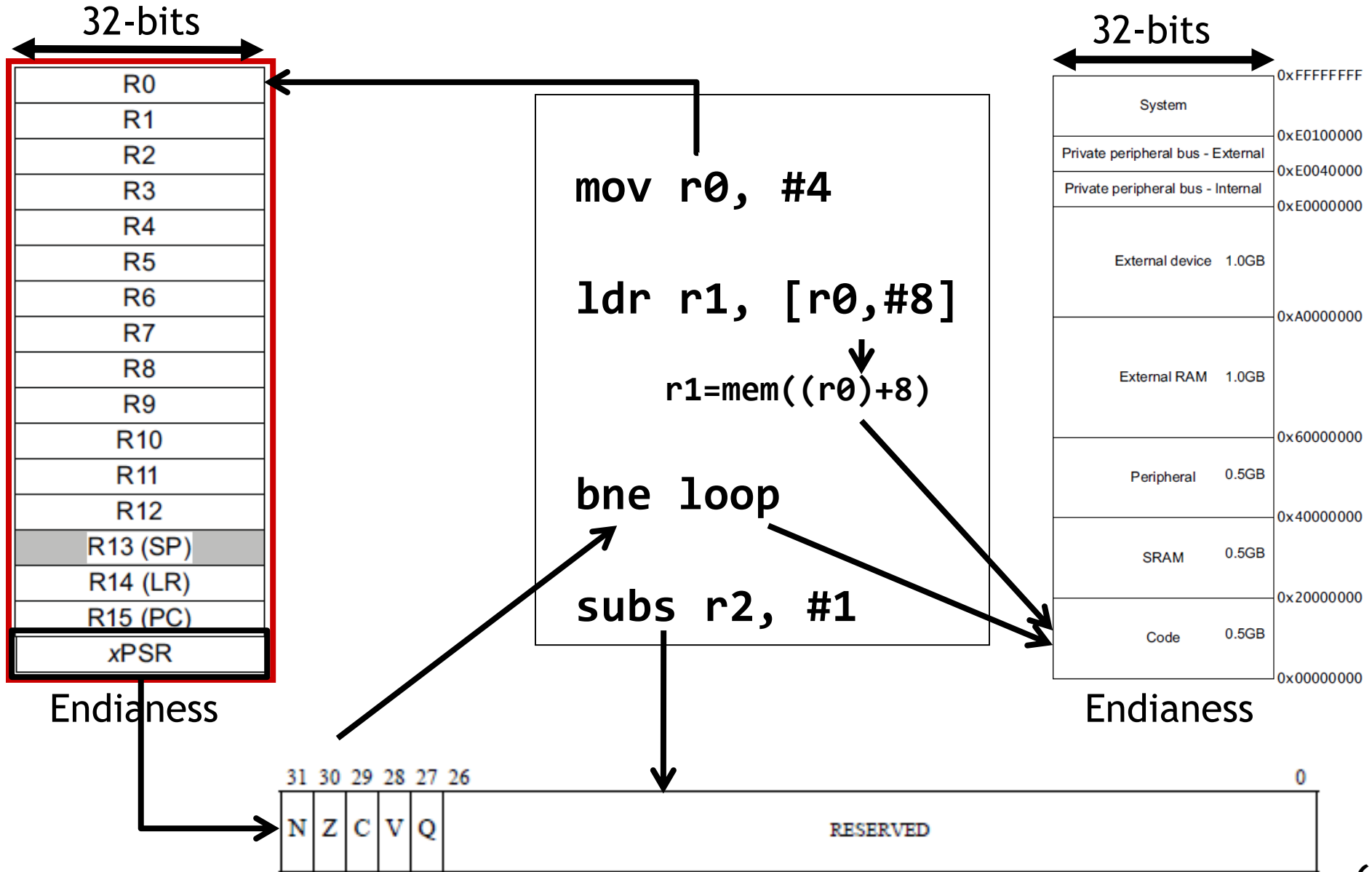
- Most defining feature of an architecture
 - IA-32 (Intel Architecture, 32-bit)
- Word size is often used as
 - 8-bit, 16-bit, 32-bit, or 64-bit machine, microcontroller, microprocessor, or computer
- Determines the size of the addressable memory
 - A 32-bit machine can address 2^{32} bytes
 - 2^{32} bytes = 4,294,967,296 bytes = 4GB
 - Note: just because you can address, it doesn't mean that there's actually something there!
- Embedded systems use still 8/16/32 bits
 - Code density/size/expressiveness
 - CPU performance/addressable memory

Word size - ARM and Thumb Instruction Set

- Early ARM instruction set
 - **32 bit** instruction set, called ARM instructions
 - Powerful and good performance
 - Larger program memory compared to 8-bit and 16-bit processors
 - Large power consumption
- **Thumb-1** instruction set
 - **16 bit** instruction set , provides a subset of the ARM instructions
 - Degraded performance
 - Smaller program memory consumed due to better code density
 - Code size reduced by ~30% whereas performance is degraded by ~20%

Introduction to ARM ISA

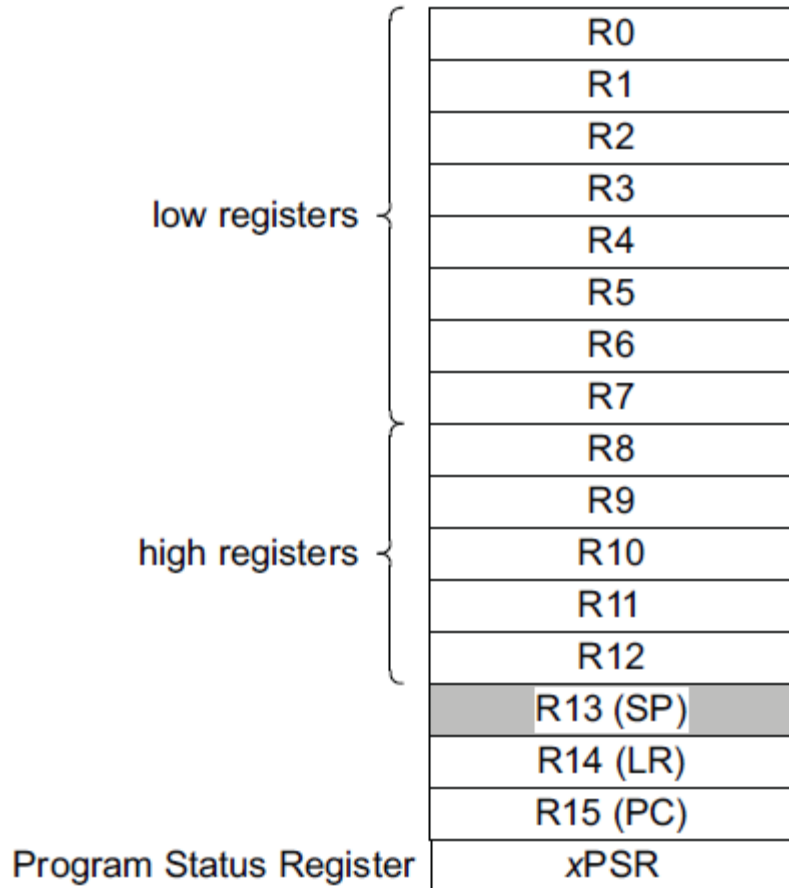
(word size, **registers**, memory, endianness, conditions, instructions, addressing modes)



ARM Registers

- ARM has **37** registers in total, all of which are **32-bits** long.
 - 1 dedicated program counter
 - 1 dedicated current program status register
 - 5 dedicated saved program status registers
 - 30 general purpose registers
- However the accessible registers are governed by the processor mode. Each mode can access
 - R0 - R15 (explained more in later slide)
 - cpsr (the current program status register)
- And privileged mode can access
 - a particular spsr (saved program status register)

ARM Cortex-M3 Registers



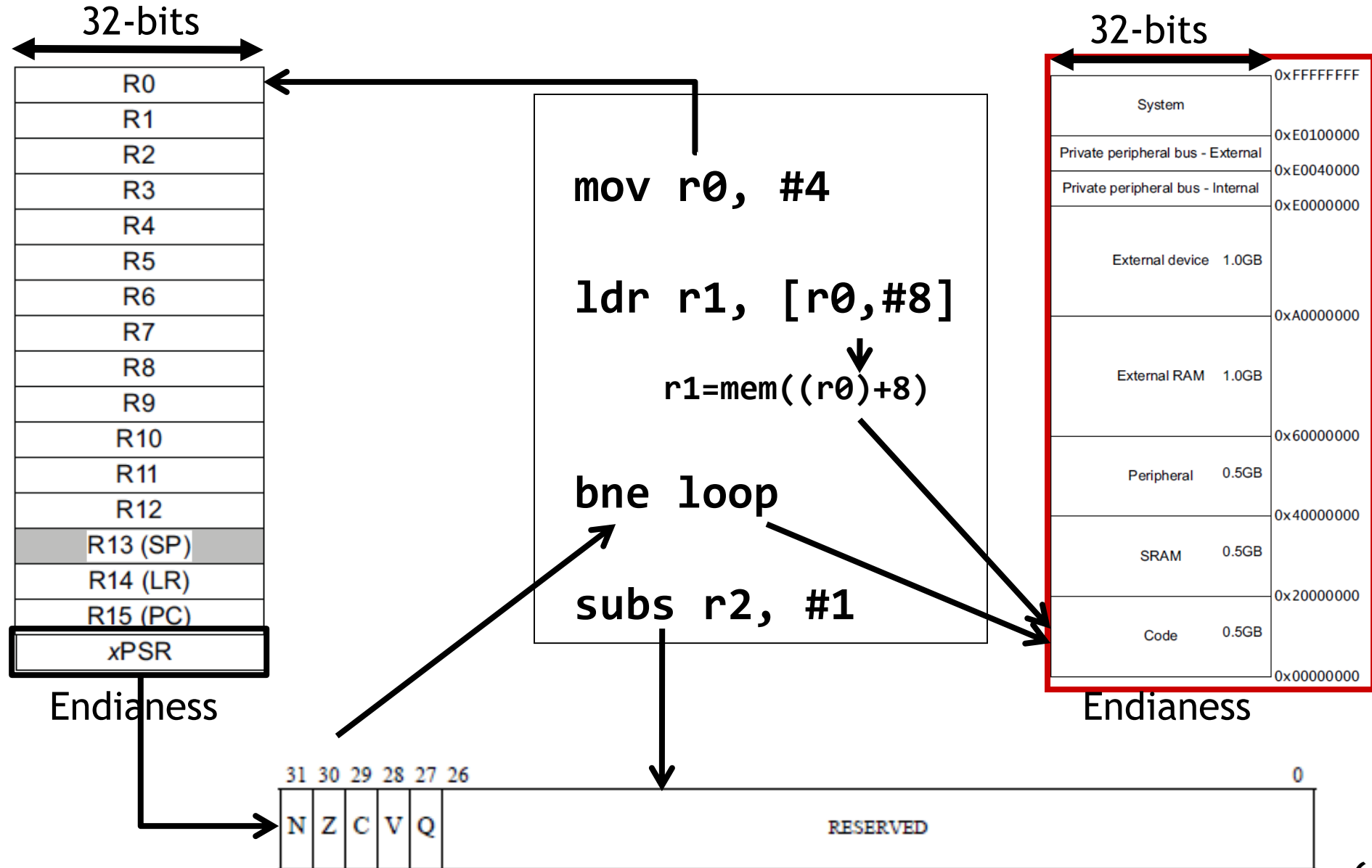
- R0-R12
 - General-purpose registers
 - Some 16-bit (Thumb) instruction only access R0-R7
- R13 (SP, PSP, MSP)
 - Stack pointer(s)
 - More details on next slide
- R14 (LR)
 - Link Register
 - When a subroutine is called, return address kept in LR
- R15 (PC)
 - Holds the currently executing program address
 - Can be written to control program flow

ARM Cortex-M3 Registers

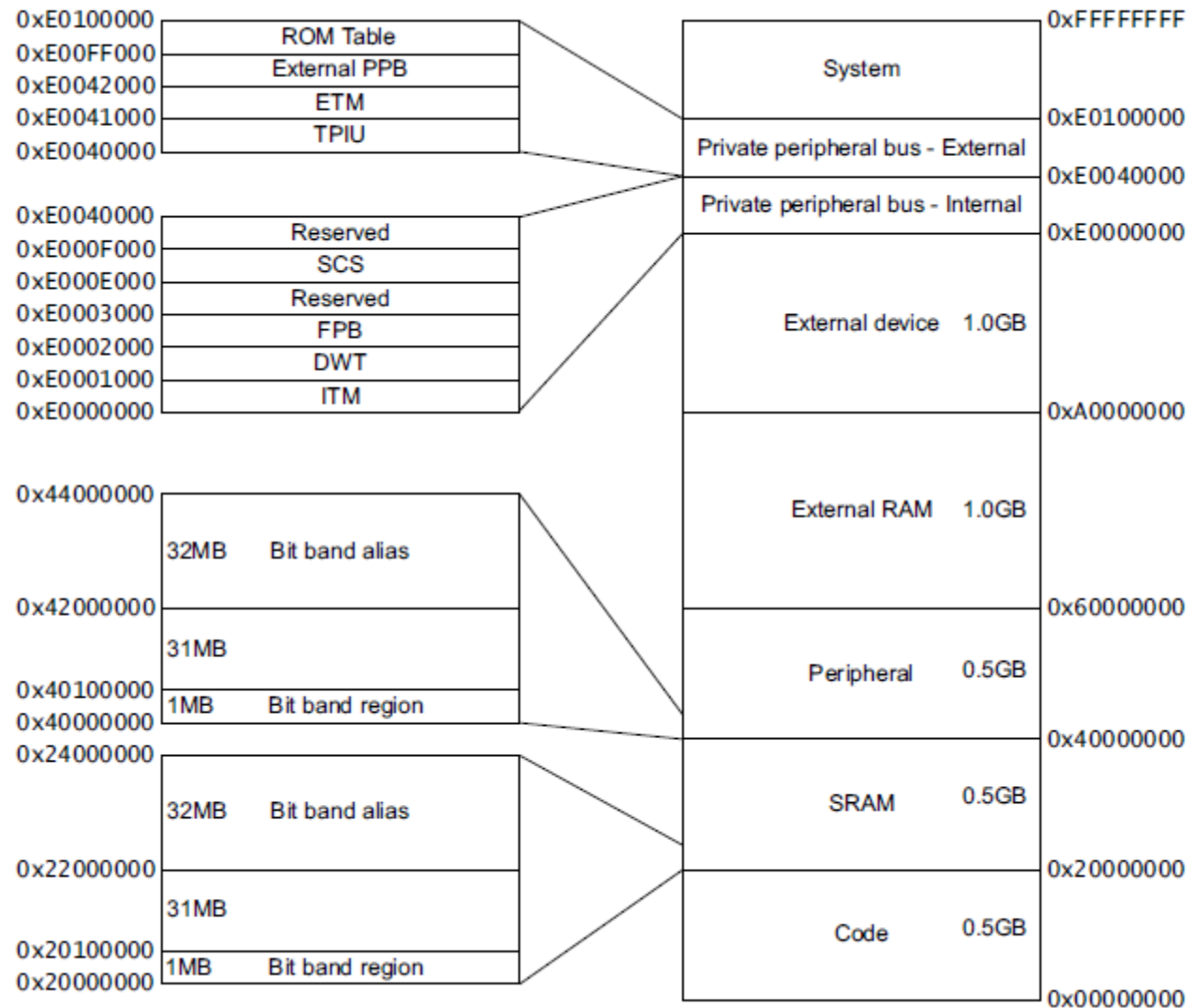
- xPSR
 - Program Status Register
 - Provides arithmetic and logic processing flags
- PRIMASK, FAULTMASK, BASEPRI
 - Interrupt mask registers
 - PRIMASK: disable all interrupts except NMI and hard fault
 - FAULTMASK: disable all interrupts except NMI
 - BASEPRI: Disable all interrupts of specific priority level or lower
- CONTROL
 - Control register
 - Define privileged status and stack pointer selection

Introduction to ARM ISA

(word size, registers, memory, endianness, conditions, instructions, addressing modes)

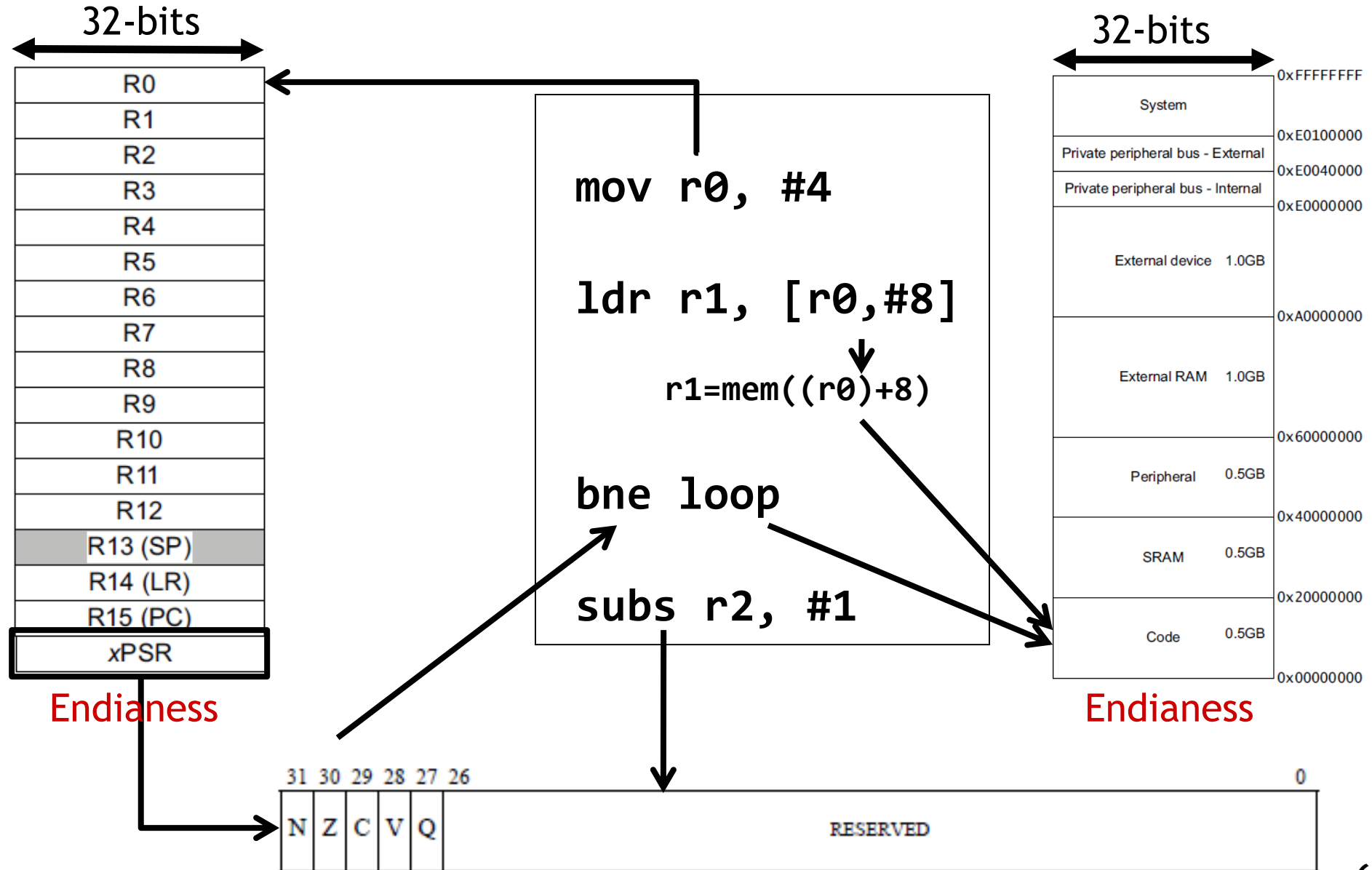


ARM Cortex-M3 Address Space / Memory Map



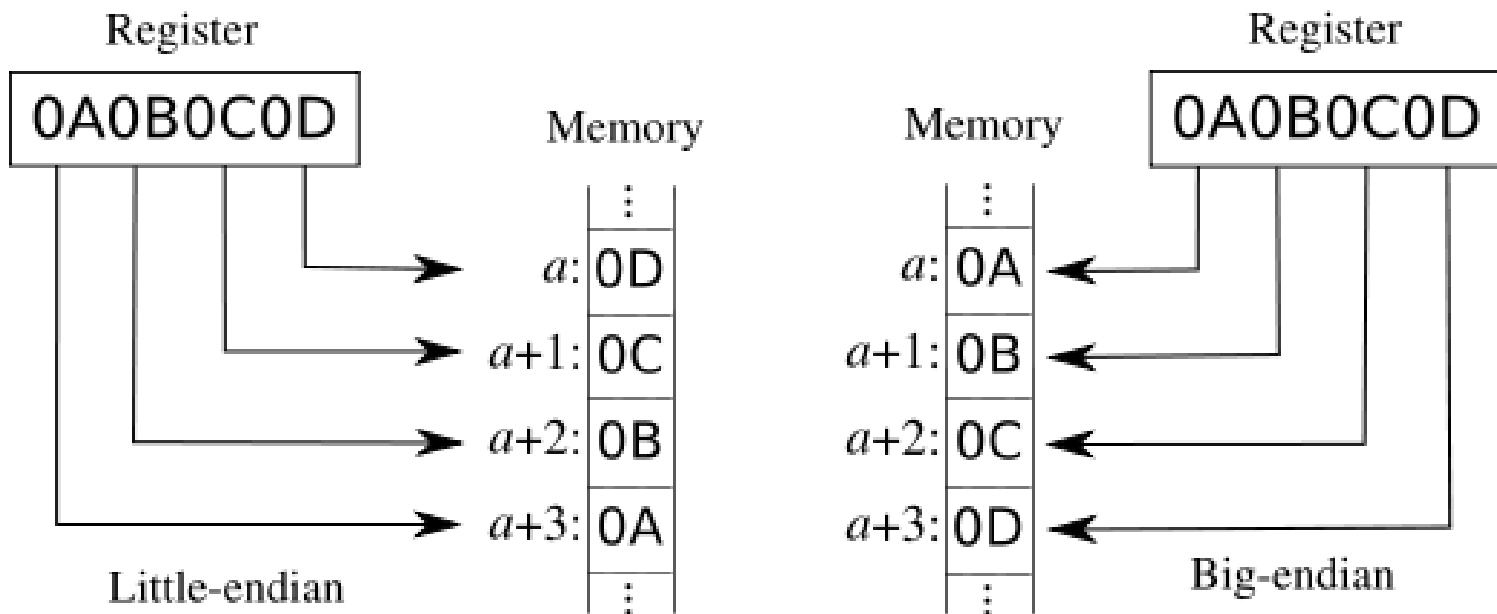
Introduction to ARM ISA

(word size, registers, memory, endianess, conditions, instructions, addressing modes)



Addressing: Big Endian vs Little Endian

- Endian-ness: ordering of bytes within a word
 - Little - increasing numeric significance with increasing memory addresses
 - Big - The opposite, most significant byte first
 - MIPS is big endian, x86 is little endian

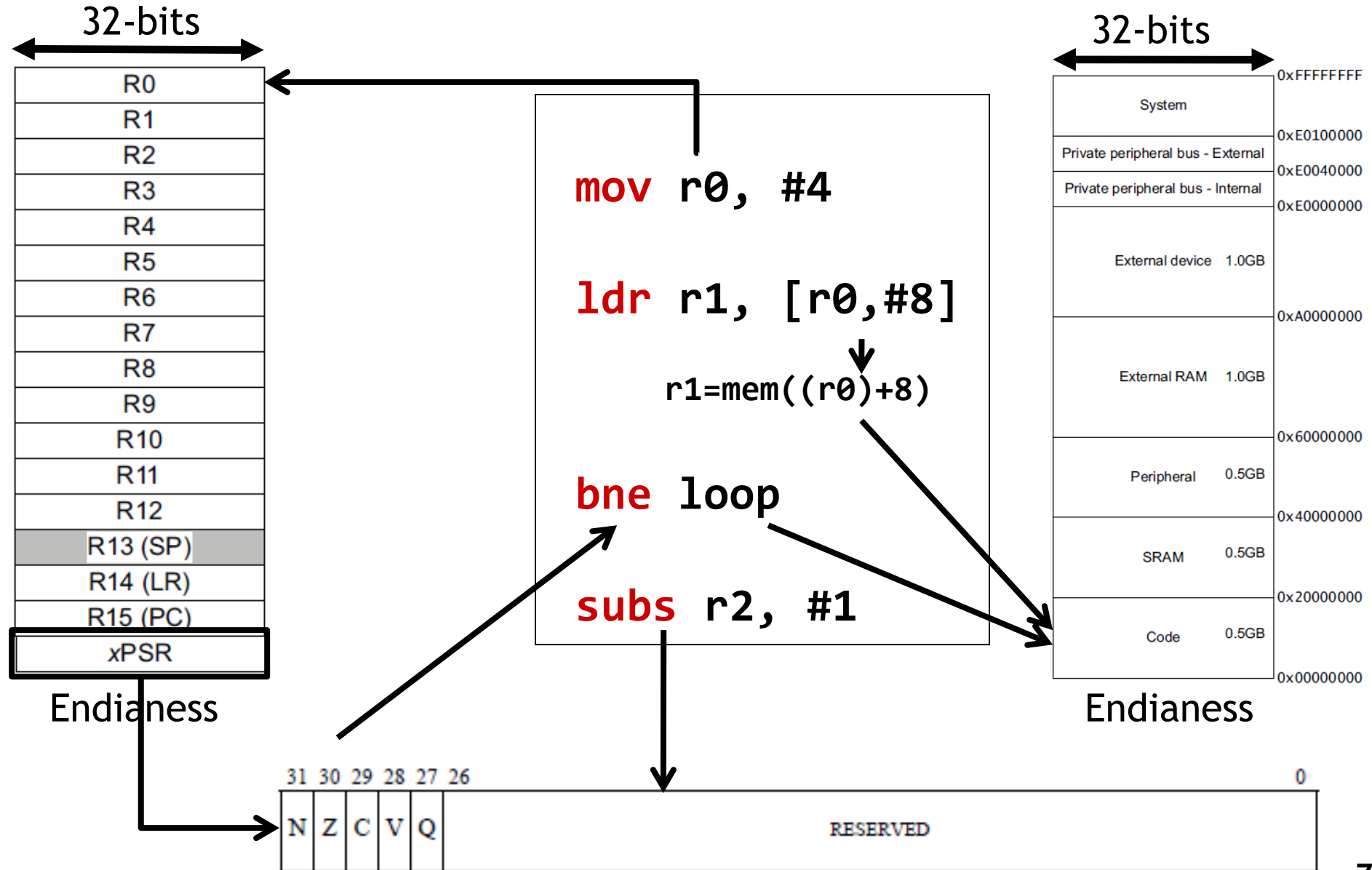


ARM Cortex-M3 Memory Formats (Endian)

- Default memory format for ARM CPUs: LITTLE ENDIAN
- Bytes 0-3 hold the first stored word
- Bytes 4-7 hold the second stored word
- Processor contains a configuration pin **BIGEND**
 - Enables hardware system developer to select format:
 - Little Endian
 - Big Endian (BE-8)
 - Pin is sampled on reset
 - Cannot change endianness when out of reset
- Source: [ARM TRM] ARM DDI 0337E, “Cortex-M3 Technical Reference Manual,” Revision r1p1, pg 67 (2-11).

Introduction to ARM ISA

(word size, registers, memory, endianness, conditions, instructions, addressing modes)



Instruction encoding

- Instructions are encoded in machine language opcodes
- Sometimes
 - Necessary to hand generate opcodes
 - Necessary to verify assembled code is correct

Instructions

movs r0, #10

movs r1, #0

Register Value

001|00|000|00001010

(msb)

(lsb)

Memory Value

(LSB) (MSB)

0a 20 00 21

001|00|001|00000000

ARMv7 ARM

Encoding T1

All versions of the Thumb ISA.

MOVS <Rd>, #<imm8>

MOV<C> <Rd>, #<imm8>

Outside IT block.

Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Rd			imm8							

d = UInt(Rd); setflags = !InITBlock(); imm32 = ZeroExtend(imm8, 32); carry = APSR.C;

Data processing instructions

Table A4-2 Standard data-processing instructions

Mnemonic	Instruction	Notes
ADC	Add with Carry	-
ADD	Add	Thumb permits use of a modified immediate constant or a zero-extended 12-bit immediate constant.
ADR	Form PC-relative Address	First operand is the PC. Second operand is an immediate constant. Thumb supports a zero-extended 12-bit immediate constant. Operation is an addition or a subtraction.
AND	Bitwise AND	-
BIC	Bitwise Bit Clear	-
CMN	Compare Negative	Sets flags. Like ADD but with no destination register.
CMP	Compare	Sets flags. Like SUB but with no destination register.
EOR	Bitwise Exclusive OR	-
MOV	Copies operand to destination	Has only one operand, with the same options as the second operand in most of these instructions. If the operand is a shifted register, the instruction is an LSL, LSR, ASR, or ROR instruction instead. See <i>Shift instructions</i> on page A4-10 for details. Thumb permits use of a modified immediate constant or a zero-extended 16-bit immediate constant.

Many, Many More!

Data processing instructions

- Consist of :

- **Arithmetic:** ADD ADC SUB SBC RSB RSC
- **Logical:** AND ORR EOR BIC
- Comparisons: CMP CMN TST TEQ
- Data transfer: MOV MVN

- Remember, **only LOAD/STORE can access memory.**

- All other instructions only work **on registers**, not memory

- They perform a specific operation on one or two operands

- First operand is always a register
- Second operand sent to the ALU via barrel shifter

Example : ADD immediate

Encoding T1 All versions of the Thumb ISA.

ADD{S} <Rd>, <Rn>, #<imm3>

ADD{C} <Rd>, <Rn>, #<imm3>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	imm3			Rn			Rd		

Encoding T2 All versions of the Thumb ISA.

ADD{S} <Rdn>, #<imm8>

ADD{C} <Rdn>, #<imm8>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	Rdn			imm8							

Encoding T3 ARMv7-M

ADD{S}<C> .W <Rd>, <Rn>, #<const>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	0	0	0	S	Rn			0	imm3			Rd			imm8									

Encoding T4 ARMv7-M

ADDW{C} <Rd>, <Rn>, #<imm12>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	0	0	0	0	Rn			0	imm3			Rd			imm8									

Branch

Table A4-1 Branch instructions

Instruction	Usage	Range
<i>B</i> on page A6-40	Branch to target address	+/-1 MB
<i>CBNZ</i> , <i>CBZ</i> on page A6-52	Compare and Branch on Nonzero, Compare and Branch on Zero	0-126 B
<i>BL</i> on page A6-49	Call a subroutine	+/-16 MB
<i>BLX (register)</i> on page A6-50	Call a subroutine, optionally change instruction set	Any
<i>BX</i> on page A6-51	Branch to target address, change instruction set	Any
<i>TBB</i> , <i>TBH</i> on page A6-258	Table Branch (byte offsets)	0-510 B
	Table Branch (halfword offsets)	0-131070 B

Load/Store instructions

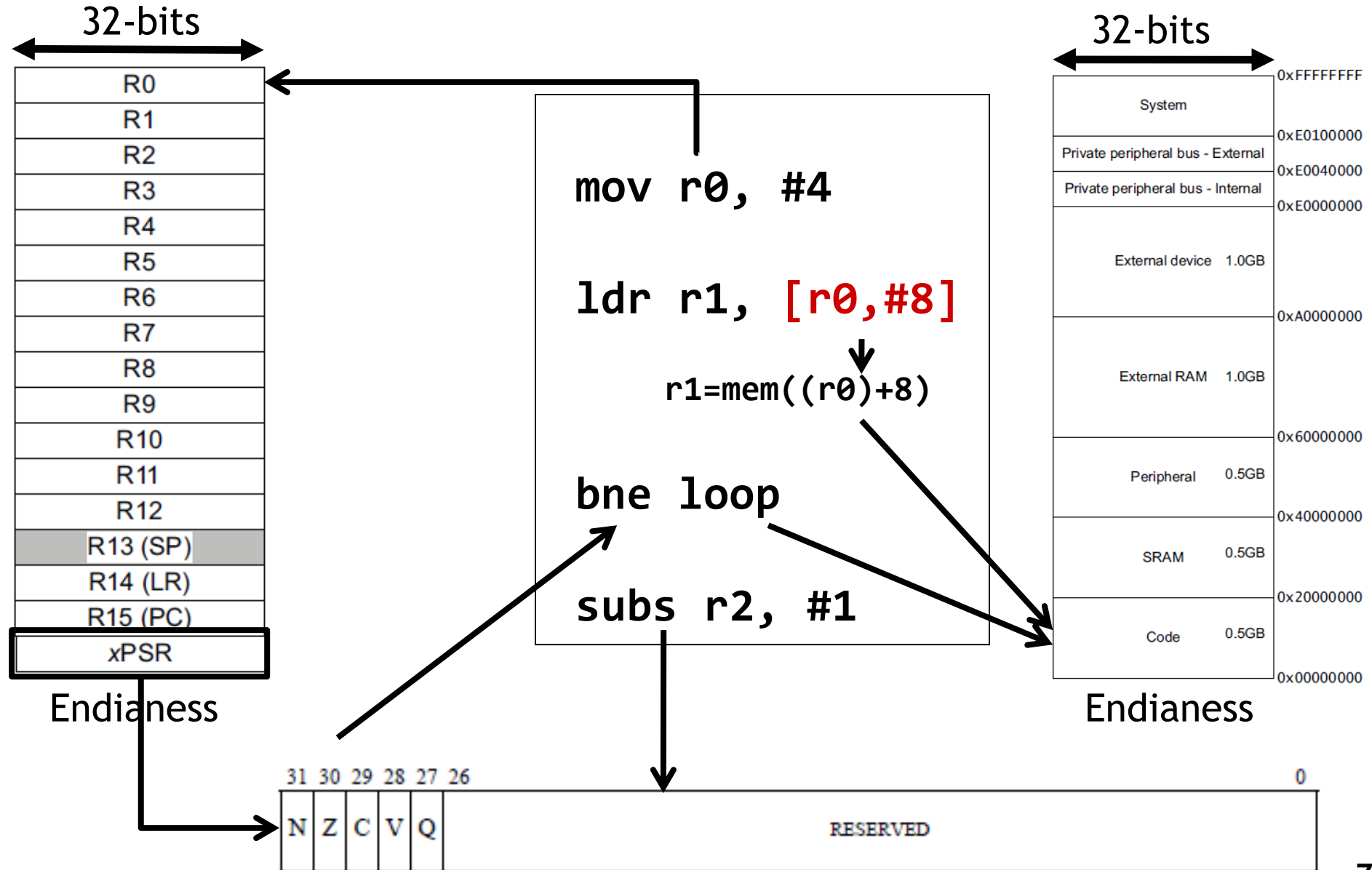
Table A4-10 Load and store instructions

Data type	Load	Store	Load unprivileged	Store unprivileged	Load exclusive	Store exclusive
32-bit word	LDR	STR	LDRT	STRT	LDREX	STREX
16-bit halfword	-	STRH	-	STRHT	-	STREXH
16-bit unsigned halfword	LDRH	-	LDRHT	-	LDREXH	-
16-bit signed halfword	LDRSH	-	LDRSHT	-	-	-
8-bit byte	-	STRB	-	STRBT	-	STREXB
8-bit unsigned byte	LDRB	-	LDRBT	-	LDREXB	-
8-bit signed byte	LDRSB	-	LDRSBT	-	-	-
two 32-bit words	LDRD	STRD	-	-	-	-

- Used to move signed and unsigned word, half word, and byte to and from registers
- Can be used to load PC
(if target address is beyond branch instruction range)

Introduction to ARM ISA

(word size, registers, memory, endianness, conditions, instructions, addressing modes)

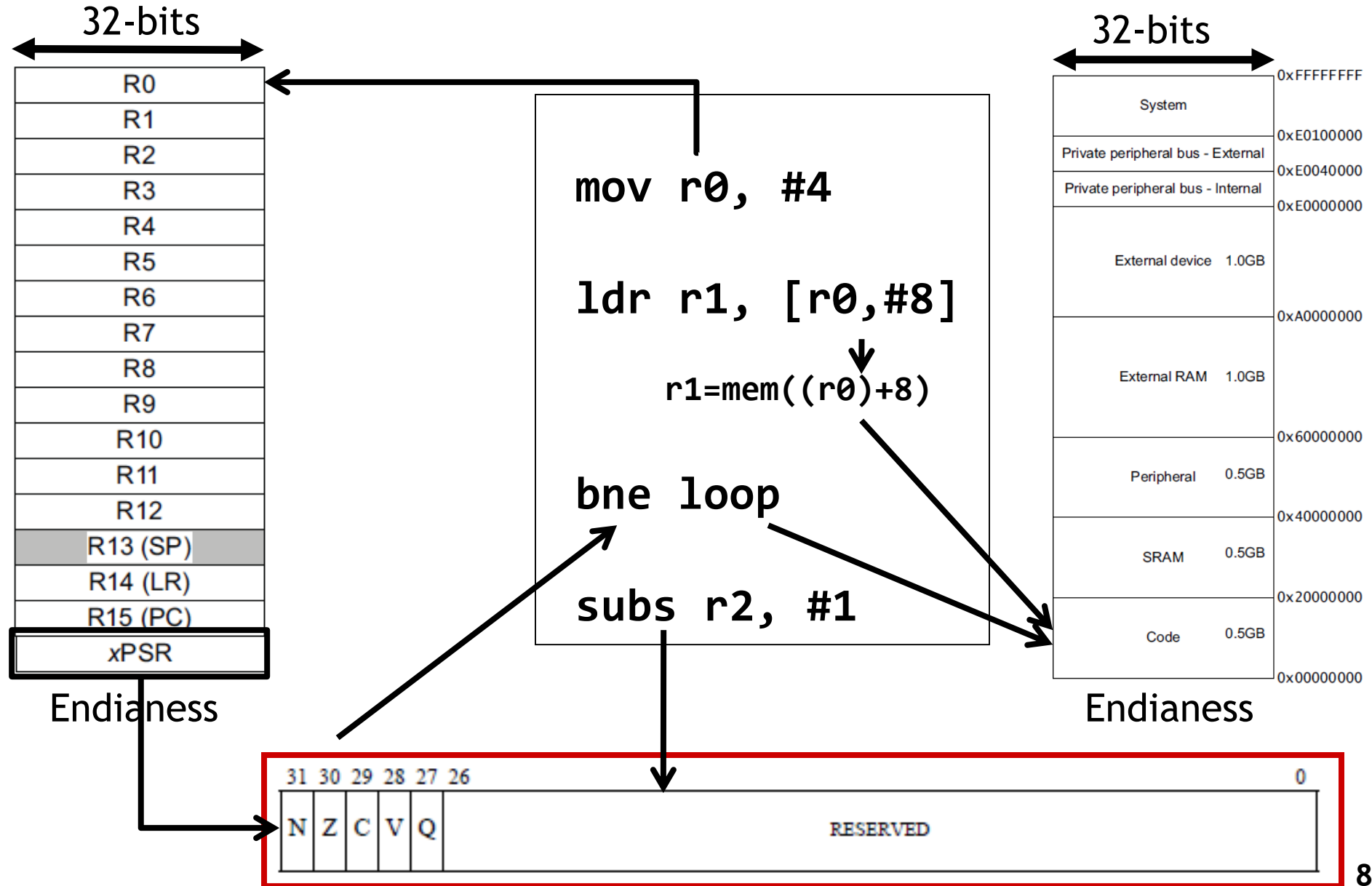


Addressing modes

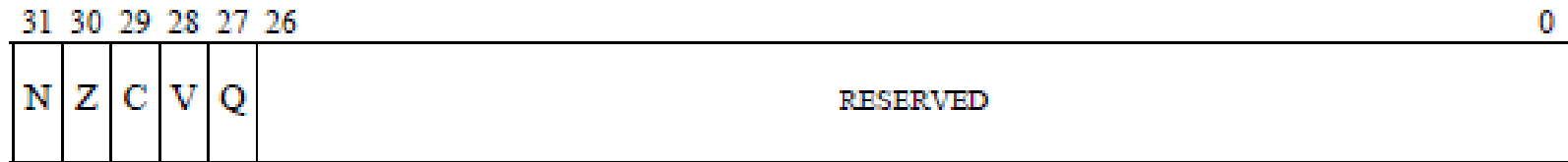
- Offset Addressing
 - Offset + or - from base register
 - Offset = immediate constant, index register, shifted IR, etc
 - Result used as effective address (EA) for memory access
 - [$\langle Rn \rangle$, $\langle \text{offset} \rangle$]
- Pre-indexed Addressing
 - Offset is applied to base register
 - Result used as effective address for memory access
 - Result written back into base register
 - [$\langle Rn \rangle$, $\langle \text{offset} \rangle$]!
- Post-indexed Addressing
 - The address from the base register is used as EA
 - The offset is applied to the base and then written back
 - [$\langle Rn \rangle$], $\langle \text{offset} \rangle$

Introduction to ARM ISA

(word size, registers, memory, endianness, conditions, instructions, addressing modes)



Application Program Status Register (APSR)



APSR bit fields are in the following two categories:

- Reserved bits are allocated to system features or are available for future expansion. Further information on currently allocated reserved bits is available in *The special-purpose program status registers (xPSR)* on page B1-8. Application level software must ignore values read from reserved bits, and preserve their value on a write. The bits are defined as UNK/SBZP.
- Flags that can be set by many instructions:
 - N, bit [31] Negative condition code flag. Set to bit [31] of the result of the instruction. If the result is regarded as a two's complement signed integer, then $N = 1$ if the result is negative and $N = 0$ if it is positive or zero.
 - Z, bit [30] Zero condition code flag. Set to 1 if the result of the instruction is zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.
 - C, bit [29] Carry condition code flag. Set to 1 if the instruction results in a carry condition, for example an unsigned overflow on an addition.
 - V, bit [28] Overflow condition code flag. Set to 1 if the instruction results in an overflow condition, for example a signed overflow on an addition.
 - Q, bit [27] Set to 1 if an SSAT or USAT instruction changes (saturates) the input value for the signed or unsigned range of the result.

Updating the APSR

- SUB Rx, Ry
 - $Rx = Rx - Ry$
 - APSR unchanged
- SUBS
 - $Rx = Rx - Ry$
 - APSR N, Z, C, V updated
- ADD Rx, Ry
 - $Rx = Rx + Ry$
 - APSR unchanged
- ADDS
 - $Rx = Rx + Ry$
 - APSR N, Z, C, V updated

Outline

1. What is a processor?

2. How can we design the processor?

- *Single purpose processor Design*
- *Optimization*
- *Architecture-level Design*
- **System-level Design**
 - **How to wrap designed processor for application?**

3. Brain Board

Operating System

- Optional software layer providing low-level services to a program (application).
 - File management, disk access
 - Keyboard/display interfacing
 - Scheduling multiple programs for execution
 - Or even just multiple threads from one program
 - Program makes system calls to the OS

```
DB file_name "out.txt" -- store file name

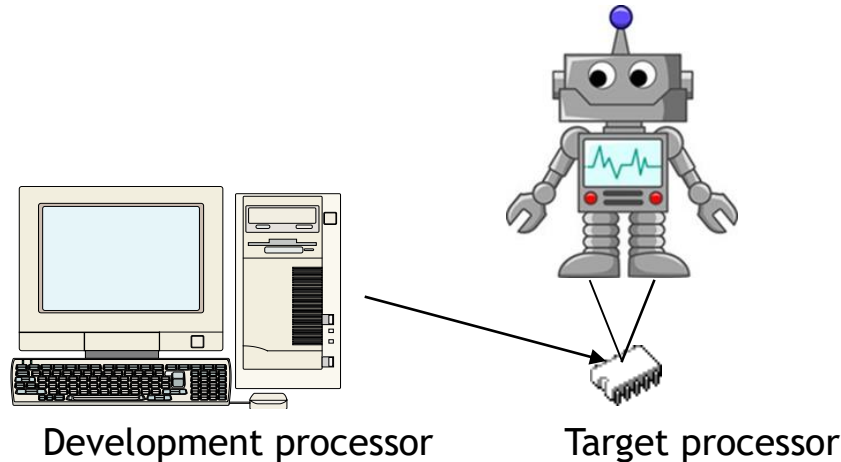
MOV R0, 1324      -- system call "open" id
MOV R1, file_name -- address of file-name
INT 34           -- cause a system call
JZ  R0, L1       -- if zero -> error

    . . . read the file
JMP L2           -- bypass error cond.
L1:
    . . . handle the error

L2:
```

Development Environment

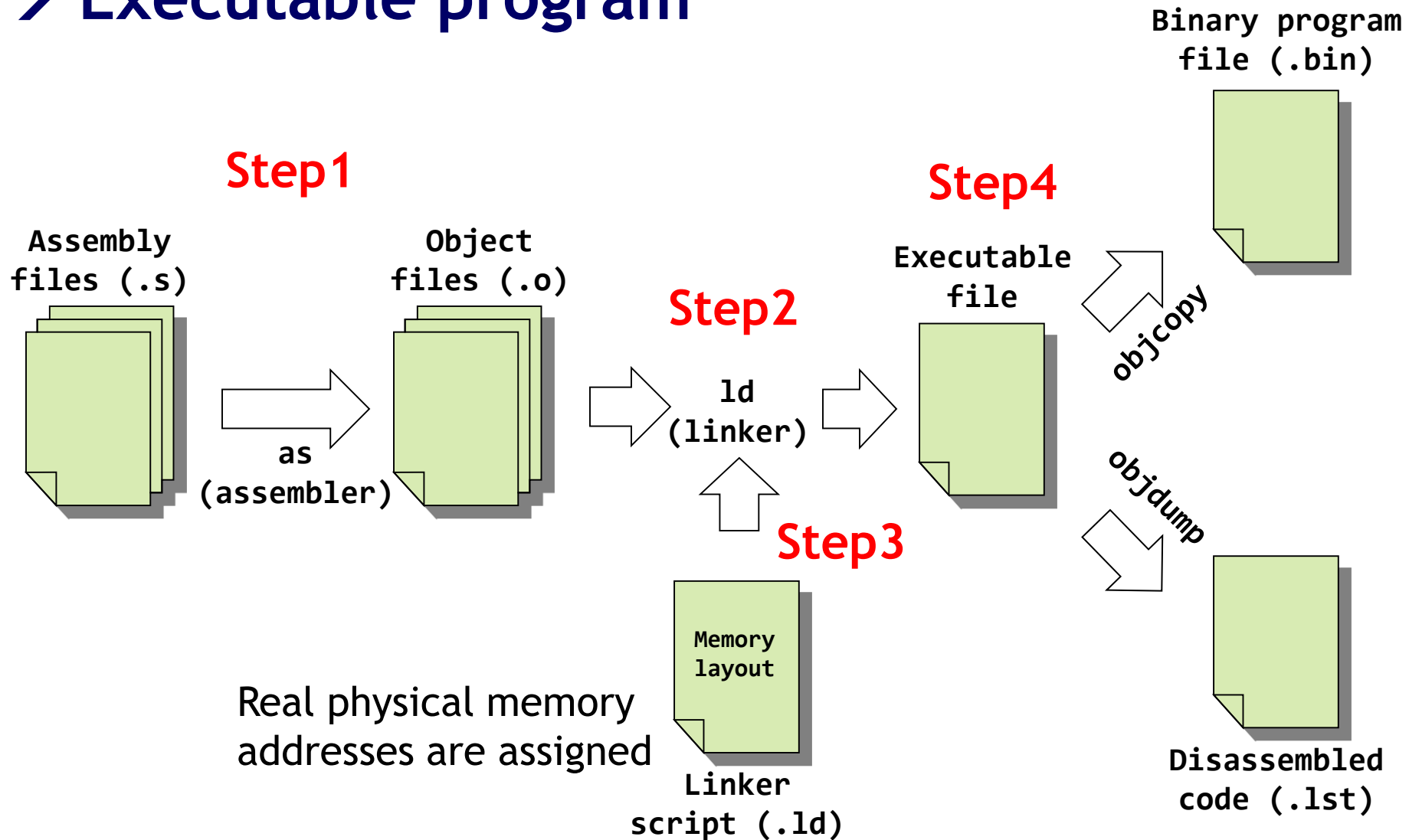
- Development processor
 - The processor on which we write and debug our programs
 - Usually a PC
- *Target processor*
 - The processor that the program will run on in our embedded system
 - Often different from the development processor



How does an assembly language program get turned into an executable program image?

- Step1. Each source files are assembled into an **object file**
- Step2. All of the object files from Step1 must be **linked** together to produce a single object file, called relocatable program
- Step3. **Physical memory addresses** must be assigned to the relative offset within the relocatable program in a process called relocation
- Step4. **Executable file** is produced and ready to run on the embedded system

Assembly language program → Executable program

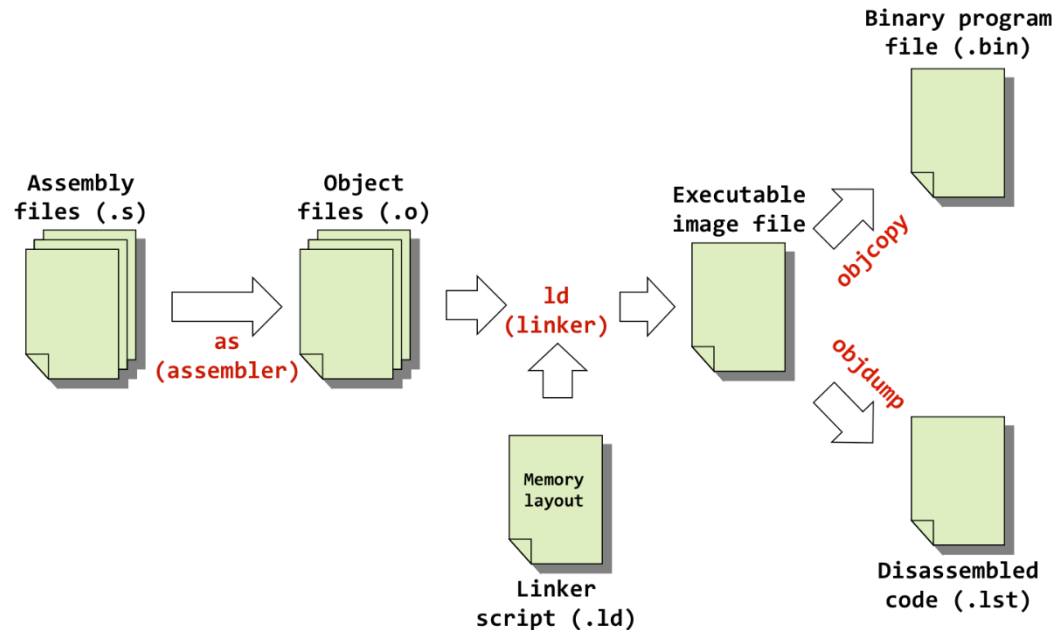


Development tools

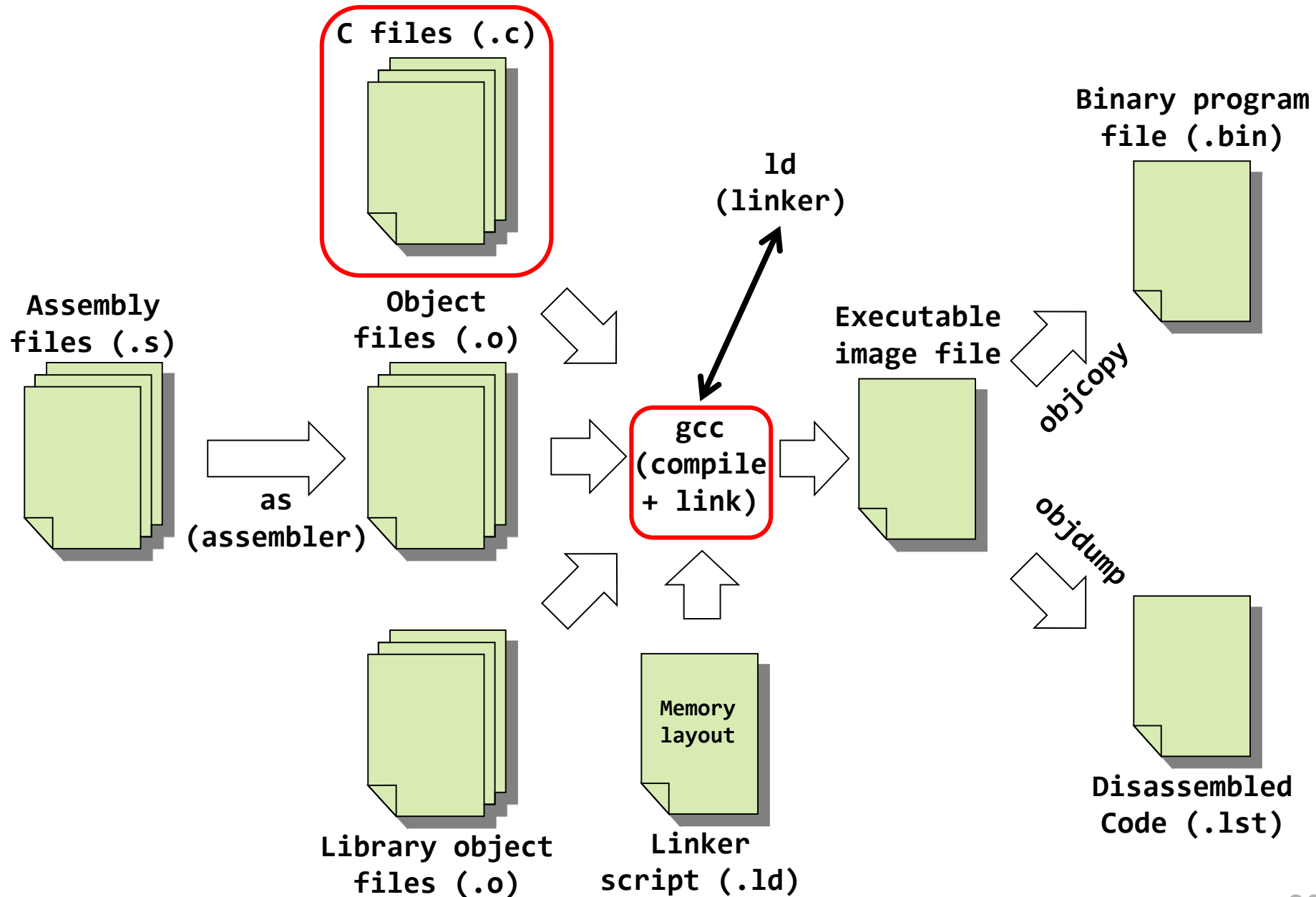
- Assembler (as)
 - Convert assembly into object file
- Linker (ld)
 - Link the object files and libraries to form an executable
- Object copy (objcopy)
 - Copies the contents of an object file to another.
 - Can convert object file into object code with another format
- Object dump (objdump)
 - Convert binary instructions into assembly one
- C/C++ Compiler (gcc, g++)
 - Translate preprocessed C(C++) into assembly

GNU executable

- Just add the prefix “arm-none-eabi-” prefix
- Assembler (as)
 - arm-none-eabi-**as**
- Linker (ld)
 - arm-none-eabi-**ld**
- Object copy (objcopy)
 - arm-none-eabi-**objcopy**
- Object dump (objdump)
 - arm-none-eabi-**objdump**
- C Compiler (gcc)
 - arm-none-eabi-**gcc**
- C++ Compiler (g++)
 - arm-none-eabi-**g++**



Mixed C/Assembly program → Executable program



Outline

1. What is a processor?

2. How can we design the processor?

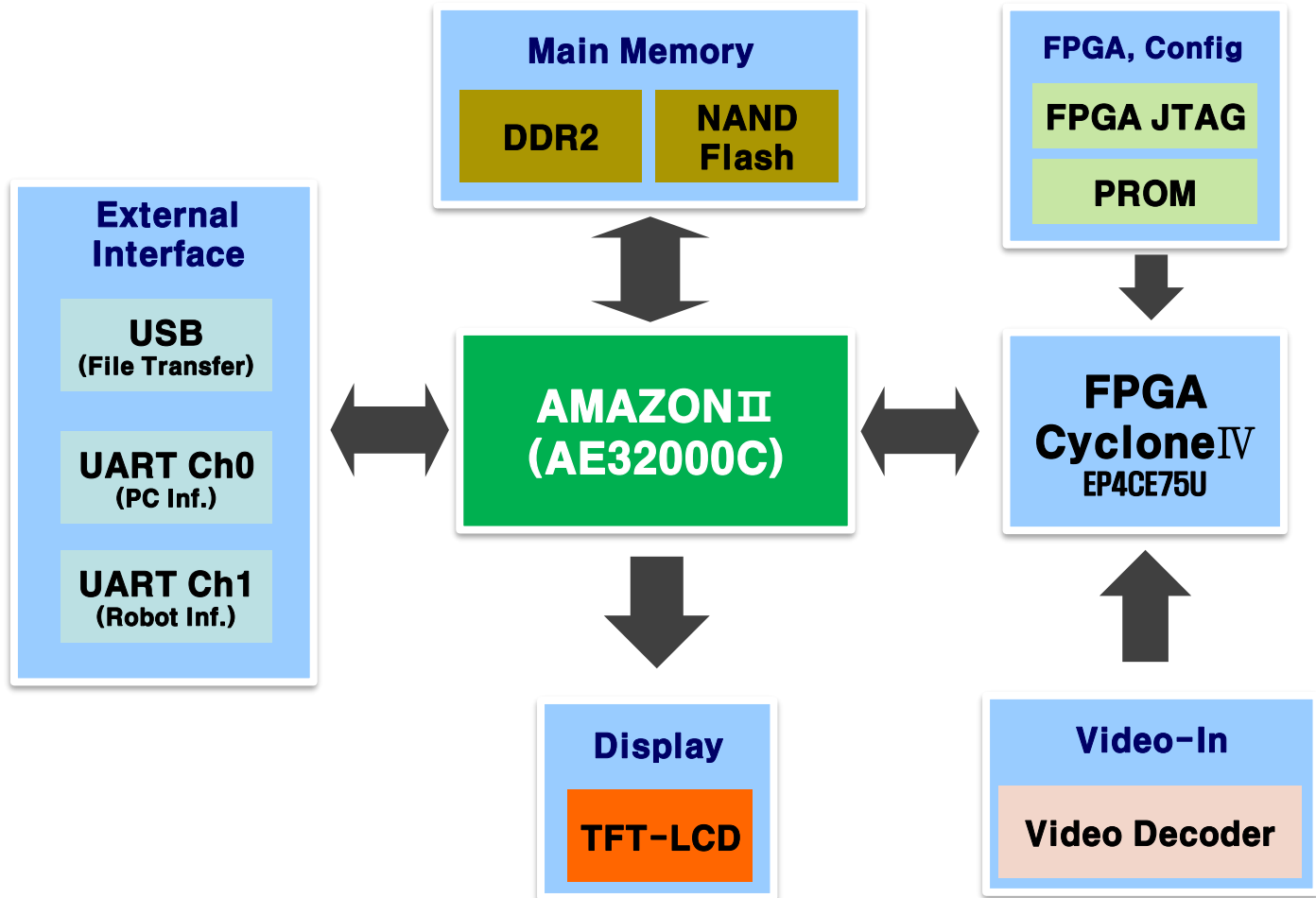
- *Combinational Logic-level Design*
- *Sequential Logic-level Design*
- *Optimization*
- *Architecture-level Design*
- *System-level Design*

3. Brain Board

Brain Board Block Diagram



Excellence in
Intelligent Robot,
Wearable Computer,
and Bio/Health!



ALTERA



Brain Board (Main Board)

SDIA

Excellence in
Intelligent Robot,
Wearable Computer,
and Bio/Health!

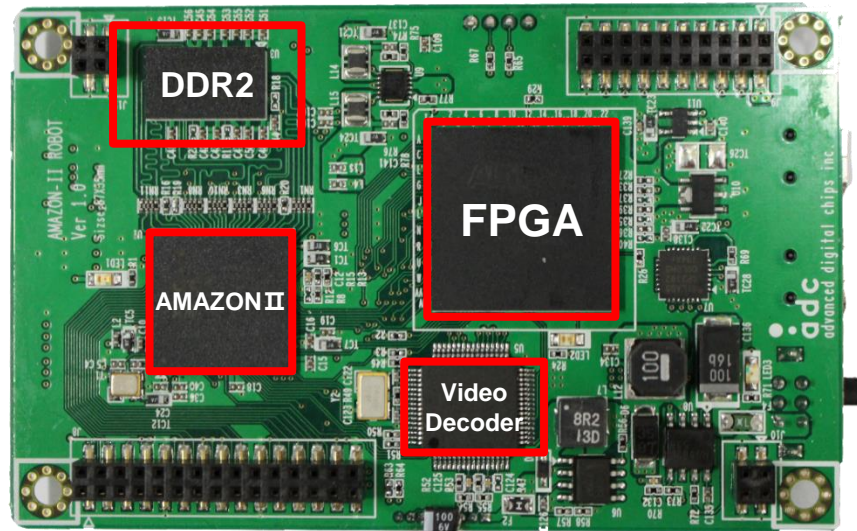
ROBOTWAR

ATERA

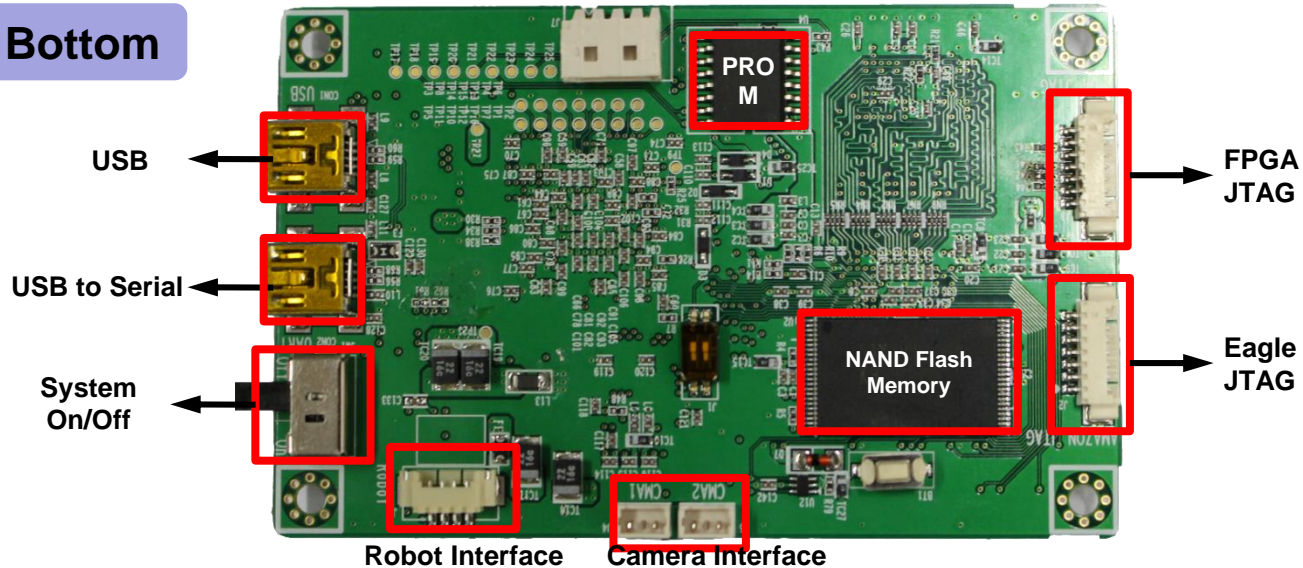
IDEC 반도체설계교육센터
IC DESIGN EDUCATION CENTER

KIRIA 한국로봇산업진흥원
KOREA INSTITUTE FOR ROBOT INDUSTRY ADVANCEMENT

Top



Bottom



SDIA

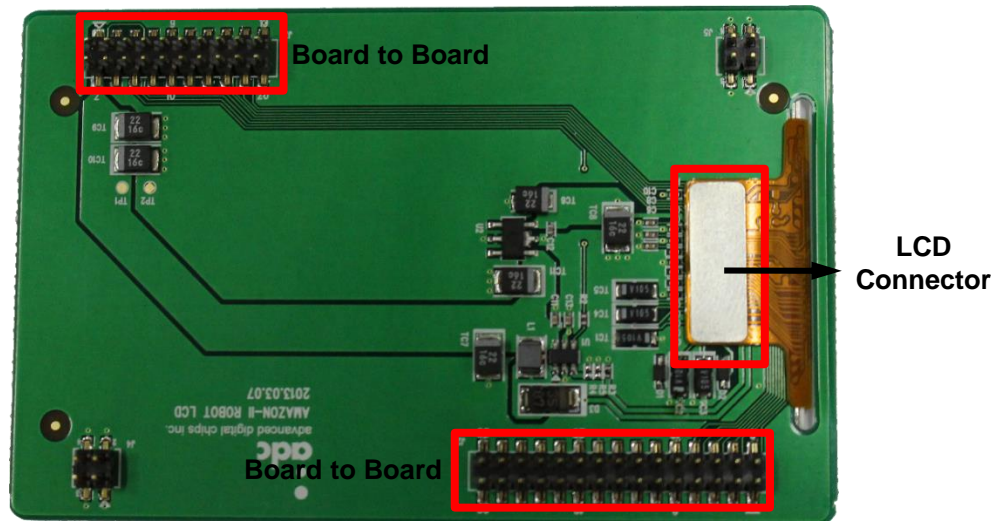
Excellence in
Intelligent Robot,
Wearable Computer,
and Bio/Health!

ROBOTWAR

Top



Bottom



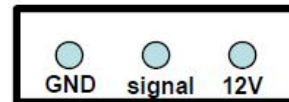
ALTERA

IDEC 반도체설계교육센터
IC DESIGN EDUCATION CENTER

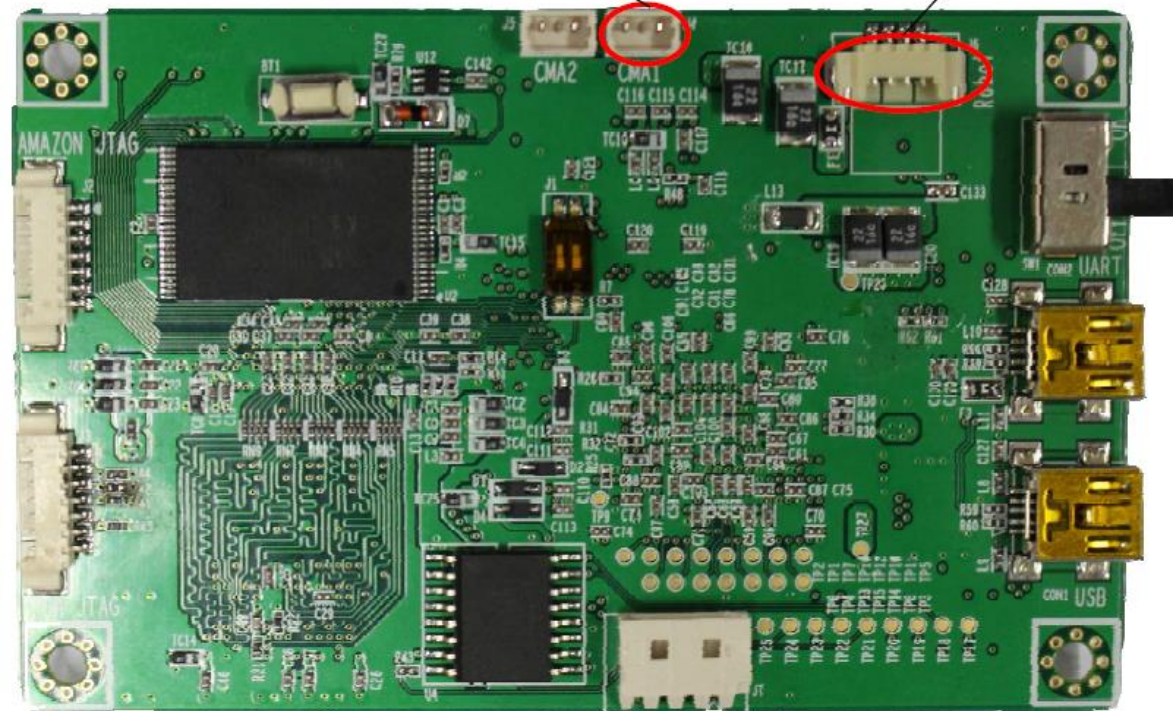
KIRIA 한국로봇산업진흥원
KOREA INSTITUTE FOR ROBOT INDUSTRY ADVANCEMENT

Connector Pin Information

Camera Interface



Robot Interface





Excellence in
Intelligent Robot,
Wearable Computer,
and Bio/Health!



Hardware

- PC
- SoC Brain Board
- USB Cable, 2ea
- DC 5V Power Adapter (2A)
- NTSC output camera (12V)

Software

- OS: Windows7 (勸獎), Windows8
- Cygwin (gcc operation environment)
- AE32000C Compiler
- USB Download Program, USB Driver
- Hyper Terminal Putty



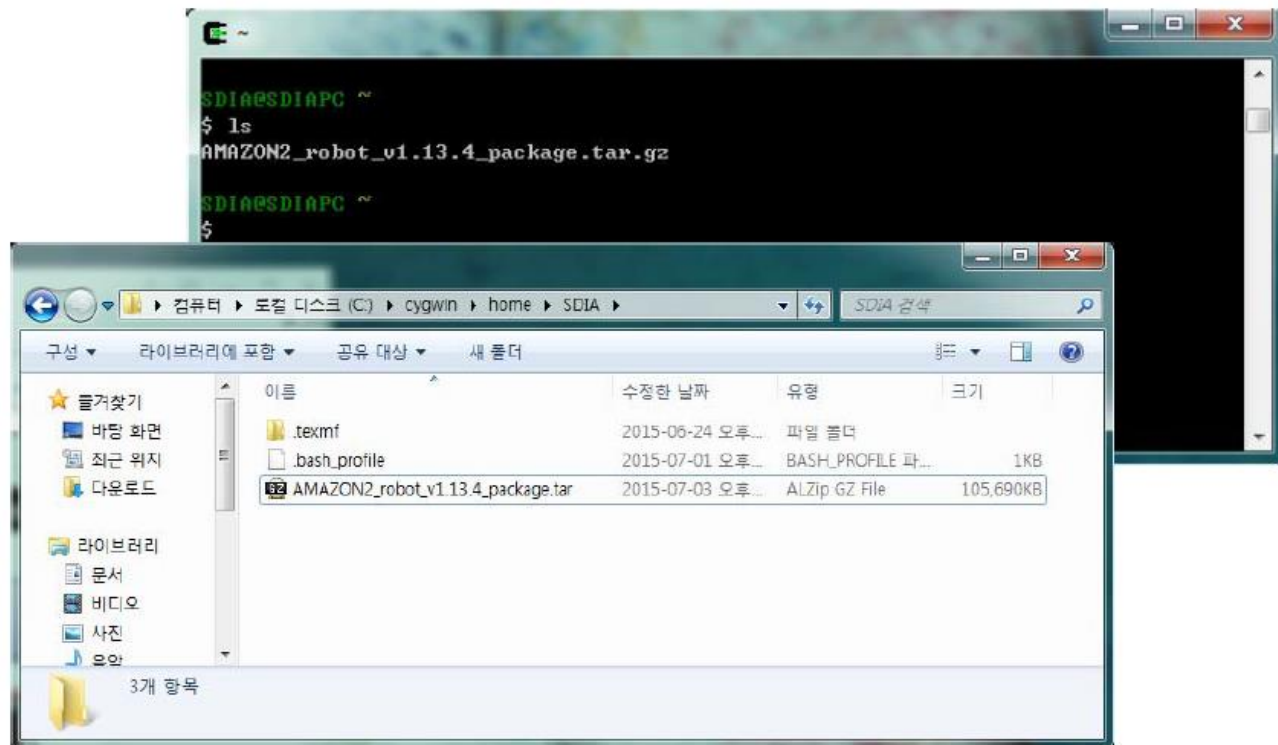


Excellence in
Intelligent Robot,
Wearable Computer,
and Bio/Health!



Compiler Installation

- AMAZON2_robot_v1.13.14_package.tar.gz
- Directory Copy: C:\cygwin\home\<user ID>





Excellence in
Intelligent Robot,
Wearable Computer,
and Bio/Health!



Linux Command

Command	Task
File/Directory Basics	
ls	List files
cp	Copy files
mv	Rename files
rm	Delete files
ln	Link files
cd	Change directory
pwd	Print current directory name
mkdir	Create directory
rmdir	Delete directory
File Viewing	

Command	Task
File Location	
find	Locate files
slocate	Locate files via index
which	Locate commands
whereis	Locate standard files
File Text Manipulation	
grep	Search text for matching lines
cut	Extract columns
paste	Append columns
tr	Translate characters
sort	Sort lines

- cat: print text files
- tar: group/ungroup files
- pwd: print the path of the current directory
- chmod: Changing authority of the file
- clear: clear the terminal
- date: print current time





Excellence in
Intelligent Robot,
Wearable Computer,
and Bio/Health!



Unzip Tar.gz

- Execute Cygwin
- `$ tar xvfz AMAZON2_robot_v1.13.4_package.tar.gz`

```
SDIA@SDIAPC ~  
$ ls  
AMAZON2_robot_v1.13.4_package.tar.gz  
  
SDIA@SDIAPC ~  
$ tar xvfz AMAZON2_robot_v1.13.4_package.tar.gz
```



Excellence in
Intelligent Robot,
Wearable Computer,
and Bio/Health!



Move to other directory

- \$ ls
- \$ cd AMAZON2_robot_v1.13.4
- \$ ls

```
~/AMAZON2_robot_v1.13.4
AMAZON2_robot_v1.13.4/user_module/LED_Module/download_usb_run.bat

SDIA@SDIAPC ~
$ ls
AMAZON2_robot_v1.13.4  AMAZON2_robot_v1.13.4_package.tar.gz

SDIA@SDIAPC ~
$ cd AMAZON2_robot_v1.13.4

SDIA@SDIAPC ~/AMAZON2_robot_v1.13.4
$ ls
kernel  toolchain  tools  user_app  user_module

SDIA@SDIAPC ~/AMAZON2_robot_v1.13.4
$
```



Excellence in
Intelligent Robot,
Wearable Computer,
and Bio/Health!



Compiler Installation

- Unzip toolchain
- \$ cd toolchain
- \$ cp ae32000-elf-uclibc-tools-AE32000C-v2.6.4.tar.gz /usr/local
- \$ cd /usr/local
- \$ tar xzf ae32000-elf-uclibc-tools-AE32000C-v2.6.4.tar.gz

```
/usr/local

SDIA@SDIAPC ~/AMAZON2_robot_v1.13.4
$ cd toolchain/

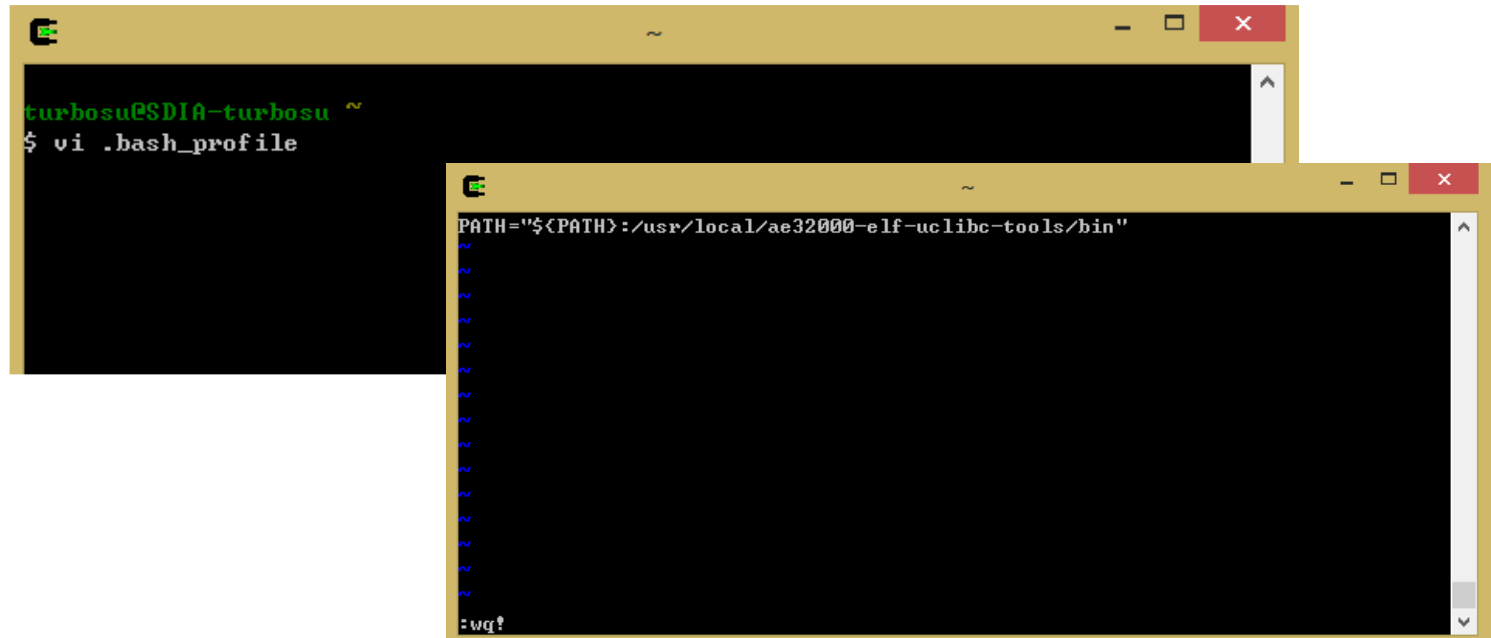
SDIA@SDIAPC ~/AMAZON2_robot_v1.13.4/toolchain
$ cp ae32000-elf-uclibc-tools-AE32000C-v2.6.4.tar.gz /usr/local/

SDIA@SDIAPC ~/AMAZON2_robot_v1.13.4/toolchain
$ cd /usr/local/

SDIA@SDIAPC /usr/local
$ tar xzf ae32000-elf-uclibc-tools-AE32000C-v2.6.4.tar.gz
```

The logo for Robot War, featuring the words "ROBOT WAR" in a bold, stylized font. "ROBOT" is in red and "WAR" is in black. To the right of the text is a grey square icon containing two dots, resembling a robot's face.

- **Make the Shell file**
- \$ vi .bash_profile
- Save after adding next PATH in vi editor
- PATH="\$PATH":/usr/local/ae32000-elf-uclibc-tools/bin"





Excellence in
Intelligent Robot,
Wearable Computer,
and Bio/Health!



Compiler Installation

- Execute the Shell & Check the installation
- `$ source .bash_profile`
- `$ ae32000-elf-uclibc-gcc -v`

```
turbosu@SDIA-turbosu ~  
$ source .bash_profile  
  
turbosu@SDIA-turbosu ~  
$ ae32000-elf-uclibc-gcc -v  
Reading specs from /usr/local/ae32000-elf-uclibc-tools/lib/gcc/ae32000-elf-uclic  
c/3.4.5/specs  
Configured with: /d/AE32000-uClinux/AE32000-uClibc/uClinux_Compiler_AE32000C_v2.  
6.4/toolchain_build_ae32000/gcc-3.4.5-ae32000c-uclibc-v080829/configure --prefix  
=/usr/local/ae32000-elf-uclibc-tools --build=i686-pc-cygwin32 --host=i686-pc-cyg  
win32 --target=ae32000-elf-uclibc --enable-languages=c,c++ --with-gxx-include-di  
r=/usr/local/ae32000-elf-uclibc-tools/ae32000-elf-uclibc/include/c++ --disable-s  
hared --disable-__cxa_atexit --enable-target-optspace --with-gnu-ld --without-pi  
c --disable-nls --enable-sjlj-exceptions  
Thread model: single  
gcc version 3.4.5 (AE32000 Compiler v2.6.4 : binutils-2.14 : gdb_insight-6.8)  
(LDI Code motion / Seperated GCCLIB / mul32 / Mem index  
 / Floating-point optimized again / Double precision BUG Fixed)  
  
turbosu@SDIA-turbosu ~  
$
```

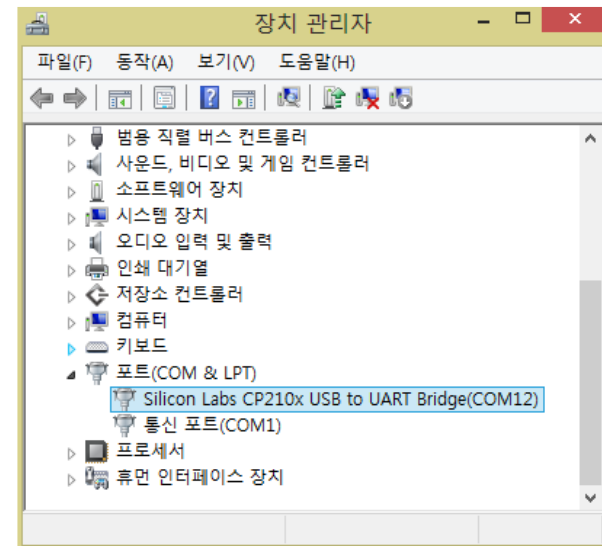
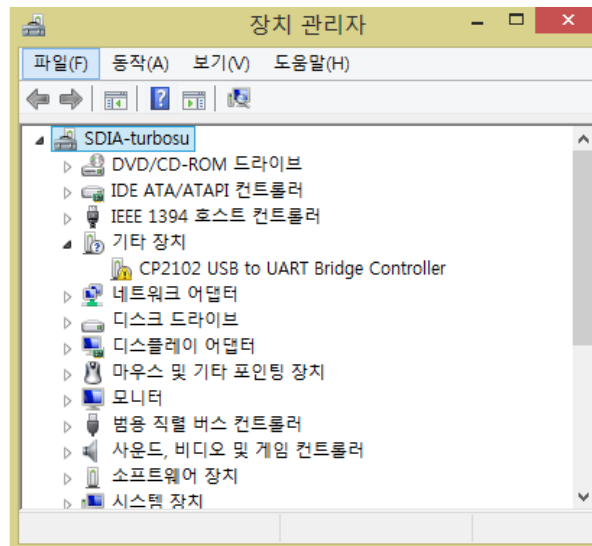


Excellence in
Intelligent Robot,
Wearable Computer,
and Bio/Health!



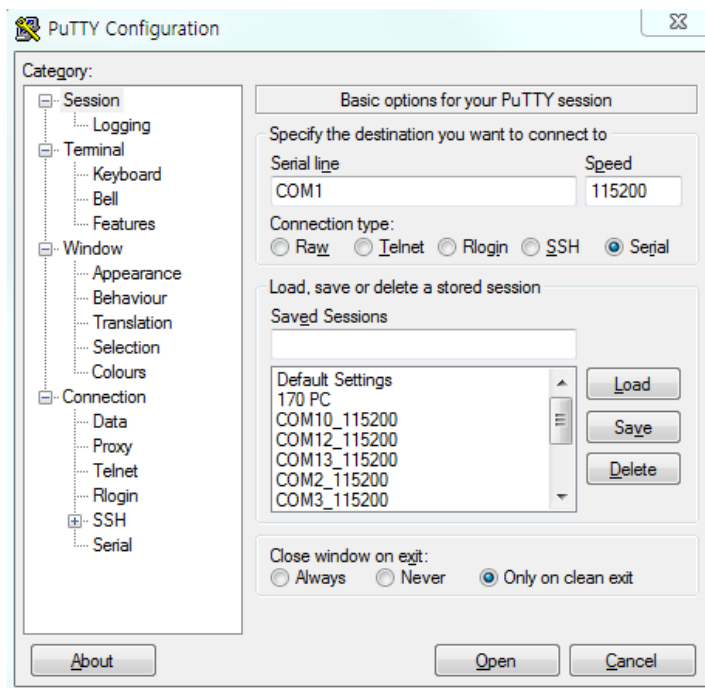
USB driver Installation

- SoC brain board connect mini USB cable to COM1, COM2
- Device Manager => CP2102 USB to UART Bridge Controller, Click
- Driver Update => Browse the driver software on your computer
- Browse => AMAZON2_robot_v1.13.4 / tools / usb_to_serial_driver select
- Click Next to install the driver.



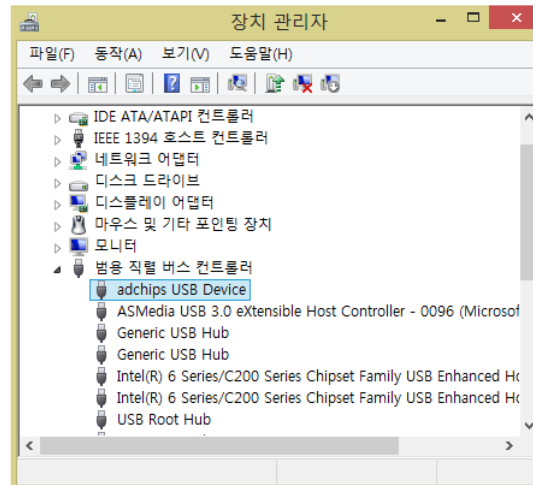
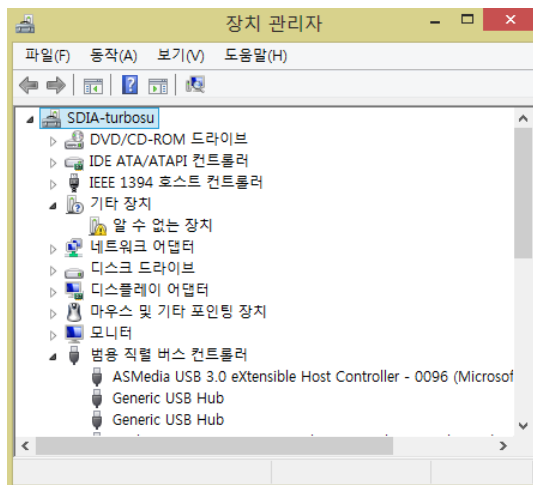
Serial Communication Program Setting

- Putty settings
- Serial Line: Communication Port
- Speed: baud rate



USB driver Installation

- Power cable connection => switch ON
- Keyboard Keystrokes from Putty (Anything)
- Device Manager => Unknown Device Click
- Driver Update => Browse the driver software on your computer
- Browse => AMAZON2_robot_v1.13.4 / tools / usb_driver select
- Click Next to install the driver.





Excellence in
Intelligent Robot,
Wearable Computer,
and Bio/Health!



Windows8 Driver Installation

Windows8 unsigned driver installation process

- Press the Windows key + I to select "Change PC settings"
- Select "General" tab, select "Restart Advanced Startup Options" button
- Select "Troubleshooting", select "Advanced Options"
- Select "Startup", "Restart" option => Reboot
- "7) Disable Driver Signature" => Keyboard 7 or F7
- Driver can be reinstalled after PC reboot

옵션 선택

→ 계속
완료 후 Windows 8으로 계속

문제 해결
PC 복구 또는 초기화, 또는 고급 도구 사용

PC 끄기

⌵ 문제 해결

PC 복구
PC가 제대로 실행되지 않는 경우 해결을 시도하거나 필요 시 PC를 복구할 수 있습니다.

PC 초기화
모든 데이터를 제거하려면 PC를 완전히 초기화합니다.

고급 옵션

⌵ 시작 설정

다시 시작하여 다음과 같은 Windows 옵션 변경:

- 저전력 모드 사용
- 디바이스 모드 사용
- 부팅 속도 향상
- 원격 모드 사용
- 드라이브 및 파일 작업 속도 향상
- 초기 실행 단계에 방지 프로그램 보호 사용 안 함
- 시스템 오류 시 자동 다시 시작 사용 안 함

다시 시작



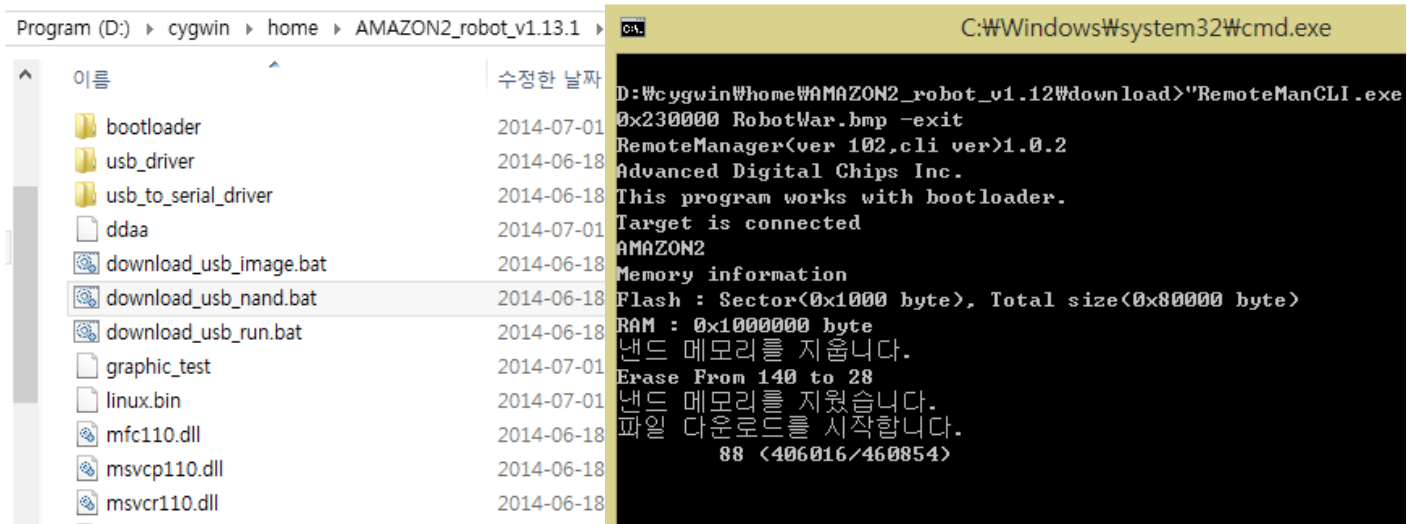


Excellence in
Intelligent Robot,
Wearable Computer,
and Bio/Health!



Kernel Ramdisk Installation

- Board power switch OFF => ON
- Enter Hyper Terminal Enter (Enter Bootloader Mode)
- File Explorer: AMAZON2_robot_v1.13.4 Move to / tools folder
- Execute download_usb_nand.bat file (Kernel, Ramdisk installation)
- Execute Download_usb_image.bat file (install boot image)





Excellence in
Intelligent Robot,
Wearable Computer,
and Bio/Health!



OS Booting

- Board power switch OFF => ON

```
socrobot UART - HyperTerminal
File Edit View Call Transfer Help
[Icons]
abled
ttyS00 at 0xf4003000 (irq = 6) is a AMAZON2 16550
AMAZON2 UART 1,2,3 deriver init done. v0.1
AMAZON2 TIMER2-3 deriver init done. v1.0
AMAZON2 LED device driver register done. v1.0
RAMDISK driver initialized: 16 RAM disks of 4096K size 1024 blocksize
NAND device: Manufacture ID: 0xec, Chip ID: 0x76 (Samsung NAND 64MB 3.3V)
Creating 1 MTD partitions on "NAND 64MB 3.3V":
0x00600000-0x04000000 : "NAND0-1"
usb.c: registered new driver hub
AMAZON2 USB BULK driver init done, Ver 1.0
RAMDISK: Compressed image found at block 0
Freeing initrd memory: 1024K
VFS: Mounted root (ext2 filesystem) readonly.
run_init_process : init_filename : /sbin/init
init started: BusyBox v1.1.0-pre1 (2008.12.22-09:57+0000) multi-call binary
EXT2-fs warning: mounting unchecked fs, running e2fsck is recommended
yaffs: dev is 7936 name is "1f:00"
yaffs: Attempting MTD mount on 31.0, "1f:00"
Using /root/module/amazon2_graphic.o
AMAZON2 Graphic Engine Driver init Done.v0.1
/mnt/f0/user_init: cannot open
ae3200c login: _
```

Connected 0:47:56 Auto detect 115200 8-N-1 SCROLL CAPS NUM Capture Print echo



Excellence in
Intelligent Robot,
Wearable Computer,
and Bio/Health!



Application SW

- Application SW Compile / Cygwin
- \$cd AMAZON2_robot_v1.13.4/user_app/hello/
- \$make clean
- \$make

```
~/AMAZON2_robot_v1.13.1/user_app/hello

turbosu@SDIA-turbosu ~
$ cd AMAZON2_robot_v1.13.1/user_app/hello/

turbosu@SDIA-turbosu ~/AMAZON2_robot_v1.13.1/user_app/hello
$ make clean
rm -f *.o *.elf *.bflt hello

turbosu@SDIA-turbosu ~/AMAZON2_robot_v1.13.1/user_app/hello
$ make
ae32000-elf-uclibc-gcc -c -Dlinux -D__linux__ -Dunix -D__unix__ -D_uClinux__ -D
EMBED hello.c -o hello.o
ae32000-elf-uclibc-gcc -r -Xlinker -T/usr/local/ae32000-elf-uclibc-tools/lib/ae3
2000-elf2flt.ld hello.o -o hello.elf
ae32000-elf-uclibc-elf2flt hello.elf
mv hello.elf.bflt hello

turbosu@SDIA-turbosu ~/AMAZON2_robot_v1.13.1/user_app/hello
$
```





Excellence in
Intelligent Robot,
Wearable Computer,
and Bio/Health!



File Download (PC => Board)

- File Explorer: AMAZON2_robot_v1.13.4 / user_app / hello
- Copy hello file, AMAZON2_robot_v1.13.4 / tools / Paste
- Hyper terminal
 - ae32000c login: root
 - #cd / mnt / f0
 - #usb_download Enter
- File Explorer: AMAZON2_robot_v1.13.4 / tools
 - Run the RemoteManCLI.exe file
 - RemoteMan <'q' to exit>> target usb
 - RemoteMan <'q' to exit>> rfw / mnt / f0 / hello hello
 - RemoteMan <'q' to exit>> run 0
 - Description: rfw / mnt / f0 / <file name to be downloaded to the board> [compiled file name]

*** If the USB driver is recognized as an unknown device,
disconnecting and reconnecting USB may help to recognize**





Excellence in
Intelligent Robot,
Wearable Computer,
and Bio/Health!



Run Application SW

- **Board**
- `#chmod 777 hello` // Change permission to run hello file
- `#./hello` // Execute the hello file

```
# chmod 777 hello
# ./hello
Hello World!!!
# _
```




Excellence in
Intelligent Robot,
Wearable Computer,
and Bio/Health!



File Upload (Board => PC)

- Hyper terminal
 - #usb_download → Enter
- File Explorer: AMAZON2_robot_v1.13.4 / tools
- Run the RemoteManCLI.exe file
 - RemoteMan <'q' to exit>> target usb
 - RemoteMan <'q' to exit>> rfr / mnt / f0 / hello new_hello
 - RemoteMan <'q' to exit>> run 0
 - Description: rfr / mnt / f0 / <file name on board> [file name to save on PC]

It is useful to capture the image from the camera, save it as a file on the board, and transfer it to the PC to analyze the image.





Excellence in
Intelligent Robot,
Wearable Computer,
and Bio/Health!



Automatically launch Application SW

- **Description:** How to start a specific program at boot time
- **Board**
 - **#vi user_init** // create user_init file and run vi editor
 - In the Vi editor, write the following phrase
 - **ESC key (command mode conversion): wq!** // Save and exit the vi editor



Excellence in
Intelligent Robot,
Wearable Computer,
and Bio/Health!



Automatically launch Application SW

- Board
 - #chmod 777 user_init
- Board power switch OFF => ON
 - Hello World !!! Check to see if

```
sgfa - HyperTerminal
File Edit View Call Transfer Help
[Icons]
abled
ttyS00 at 0xf4003000 (irq = 6) is a AMAZON2 16550
AMAZON2 UART 1,2,3 driver init done. v0.1
AMAZON2 TIMER2-3 driver init done. v1.0
AMAZON2 LED device driver register done. v1.0
RAMDISK driver initialized: 16 RAM disks of 4096K size 1024 blocksize
NAND device: Manufacture ID: 0xec. Chip ID: 0x76 (Samsung NAND 64MB 3.3V)
Creating 1 MTD partitions on "NAND 64MB 3.3V":
0x00600000-0x04000000 : "NAND0-1"
usb.c: registered new driver hub
AMAZON2 USB BULK driver init done. Ver 1.2
RAMDISK: Compressed image found at block 0
Freeing initrd memory: 1024K
VFS: Mounted root (ext2 filesystem) readonly.
run_init_process : init_filename : /sbin/init
init started: BusyBox v1.1.0-pre1 (2008.12.22-09:57+0000) multi-call binary
EXT2-fs warning: mounting unchecked fs, running e2fsck is recommended
yaffs: dev is 7936 name is "1f:00"
yaffs: Attempting MTD mount on 31.0, "1f:00"
Using /root/module/amazon2_graphic.o
AMAZON2 Graphic Engine Driver init Done.v0.3
Hello World!!!

ae32000c login: _
```



Questions?

Comments?

Discussion?