Supporting Tools for Application development

Code generation

Code debugging

Using the uP: code generation

- We define the ISA.
  - How is the application software to be developed?
  - Write in machine language? ---- No way.

  - Minimum is to provide an assembler
    - What does it takes to develop an assembler to go with the new uP?
    - We will find out .....

  - More acceptable is to provide a C compiler
    - What does it takes to develop a C compiler to go with the new uP?
    - We will find out .....

Using the uP: debug software

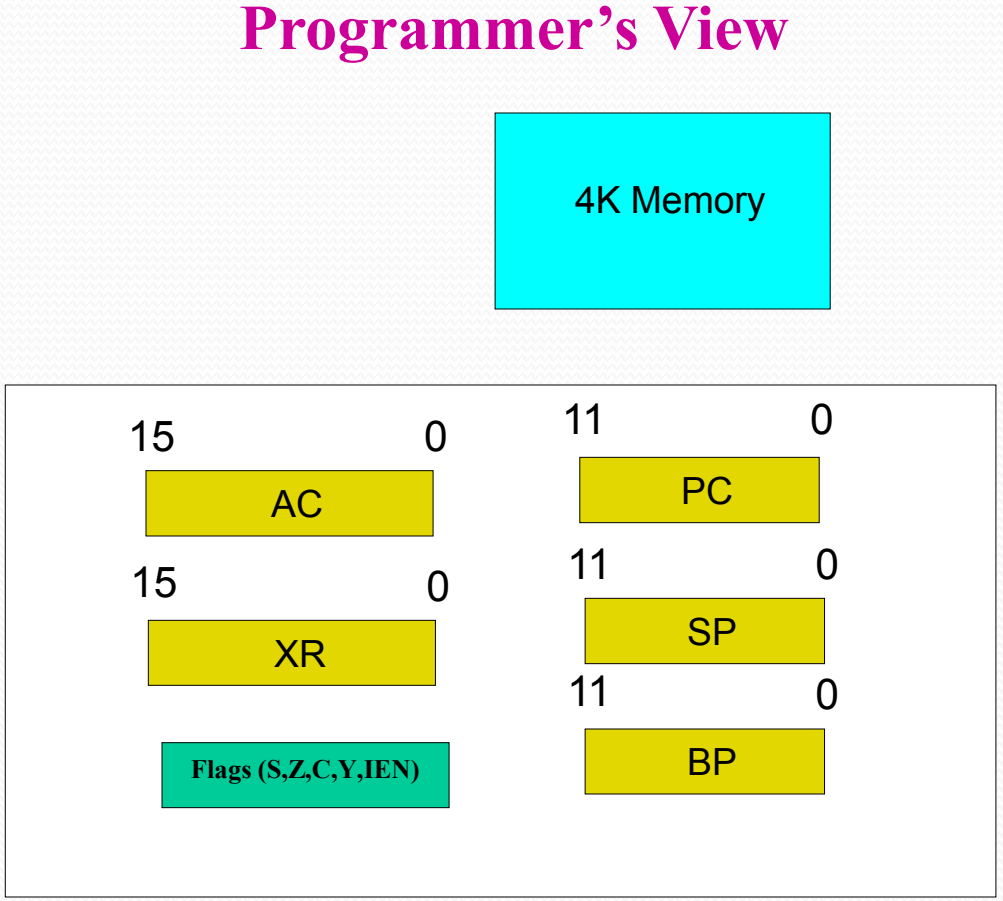
- **Most complexities are implemented in SOFTWARE**
  - **How to support software code debugging to ensure quick turnaround**
- System on a board: standalone microprocessor and other peripherals
  - Most signals are accessible
    - Hardware engineer designs board and makes PCB
      - Use oscilloscope and logic analyzer to measure signal on board
    - Software engineer writes application software
      - Use emulator,
      - For difficult situations, use logic analyzer to capture signals as program runs.
      - Software means: simulator or a software debug monitor *(recall timer.asm of EE3202)*
- System on a chip: processor is deeply embedded with other peripherals on a die.
  - No access to signals: How to debug.
  - Hardware engineer use simulation technique
  - Software engineer:
    - **On-chip emulator**
    - simulator

To be provided by uP designer !

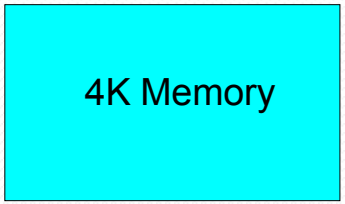
Code generation

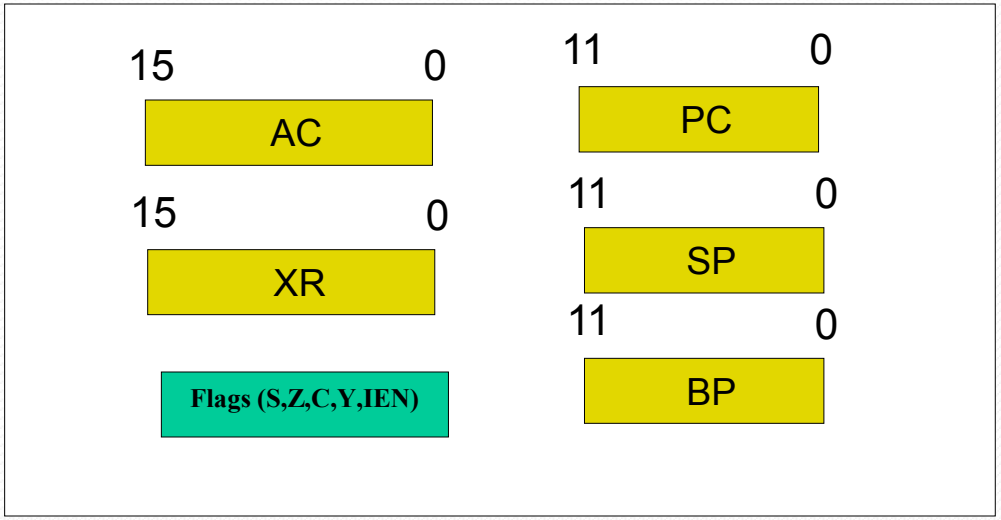
ISA: Registers and memory overview

- PC: program counter
  - Holds address of the next instruction
- SP: stack pointer
- BP: base pointer
- AC: accumulator
  - main processor register
- XR: operand register
  - For holding the operand of arithmetic/logic operations
- Flags
  - S=AC negative
  - Z=AC zero
  - C=carry flag set
  - Y=DR=o



ISA





	Hexadecimal code		
Symbol	I = 0	I = 1	Description
LDB	0xxx	8xxx	Load memory word to BP
LDX	1xxx	9xxx	Load memory word to XR
LDA	2xxx	Axxx	Load memory word to AC
STA	3xxx	Bxxx	Store content of AC in memory
BUN	4xxx	Cxxx	Branch unconditionally
CALL	5xxx	Dxxx	Push PC onto stack and branch unconditionally
DSZ	6xxx	Exxx	Decrement and skip if zero

Register reference instructions

arithmetic

logic

Shift

Bit manipulation

Subroutine /stack

Conditional branch

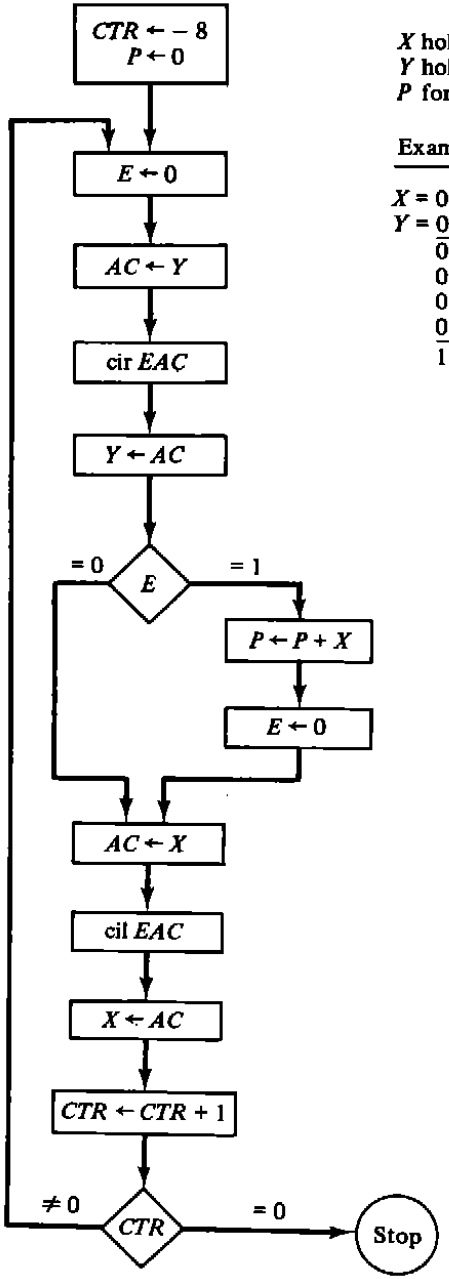
INC	7010	Increment AC
DEC	7011	Decrement AC
ADD	7012	AC=AC+XR
ADC	7013	AC=AC+XR+C
SUB	7014	AC=AC-XR
SBB	7015	AC=AC-XR+1
AND	7016	AC=AC and XR
OR	7017	AC=AC or XR
XOR	7020	AC=AC xor XR
SHR	7021	Shift AC right
SHL	7022	Shift AC left
CLAC	7023	Clear AC
CMAC	7024	Complement AC
CLC	7025	Clear C
CMC	7026	Complement C
STC	7027	Set C
AC2SP	7040	Move content of AC to SP
SP2BP	7041	Move content of SP to BP
LDABP	7042	Load AC indirect through BP
IRET	7043	Restore address from stack onto PC and ION
RET	7044	Restore address from stack onto PC
POP	7045	Pop word from the stack onto AC
PUSH	7046	Push AC onto the stack
SZAC	7047	Skip next instruction if AC zero
SNAC	7080	Skip next instruction if AC negative
SPAC	7081	Skip next instruction if AC positive
SSC	7082	Skip next instruction if C is set
SCC	7083	Skip next instruction if C is clear
ION	7084	Interrupt on
IOF	7085	Interrupt off

Assembly Language - grammar

- label: empty or a symbolic address
- instruction field
  - a memory reference instruction
    - 2 or 3 symbols separated by space
      - instruction code
      - address
      - I : direct or indirect
  - a register reference or input-output instruction
    - instruction code only
  - a pseudo instruction
    - org N: Hex N is the memory location for the instruction or operand listed in the following line
    - END
    - DEC N : signed decimal number N to be converted to binary
    - HEX N : hex N to be converted to binary
- comment

Multiply 2 positive numbers

```
        ORG 100
LOP,    CLC         /Clear E
        LDA Y       /Load multiplier
        SHR         /Transfer multiplier bit to E
        STA Y       /Store shifted multiplier
        SCC         /Check if bit is zero
        BUN ONE     /Bit is one; go to ONE
        BUN ZRO     /Bit is zero; go to ZRO
ONE,    LDX X       /Load multiplicand
        LDA P
        ADD         /Add to partial product
        STA P       /Store partial product
        CLC         /Clear E
ZRO,    LDA X       /Load multiplicand
        SHL         /Shift left
        STA X       /Store shifted multiplicand
        DSZ CTR     /Increment counter
        BUN LOP     /Counter not zero; repeat loop
        BUN OS      /Counter is zero; halt
CTR,    DEC 8       /This location serves as a counter
X,      HEX 000F    /Multiplicand stored here
Y,      HEX 000B    /Multiplier stored here
P,      HEX 0       /Product formed here
        END
```



X holds the multiplicand
Y holds the multiplier
P forms the product

Example with four significant digits

		P
X =	0000 1111	
Y =	0000 1011	0000 0000
	0000 1111	0000 1111
	0001 1110	0010 1101
	0000 0000	0010 1101
	0111 1000	1010 0101
	1010 0101	

Translated Code – hand assembled

		LOC	CODE
	ORG 100		
LOP,	CLC	100	7025
	LDA Y	101	2114
	SHR	102	7421
	STA Y	103	3114
	SCC	104	7483
	BUN ONE	105	4107
	BUN ZRO	106	410C
ONE,	LDX X	107	1113
	LDA P	108	2115
	ADD	109	7016
	STA P	10A	3115
	CLC	10B	7025
ZRO,	LDA X	10C	1113
	SHL	10D	7022
	STA X	10E	3113
	DSZ CTR	10F	6112
	BUN LOP	110	4100
	BUN OS	111	4xxx
CTR,	DEC 8	112	0008
X,	HEX 000F	113	000F
Y,	HEX 000B	114	000B
P,	HEX 0	115	0000
	END		

Refer to table of instruction codes

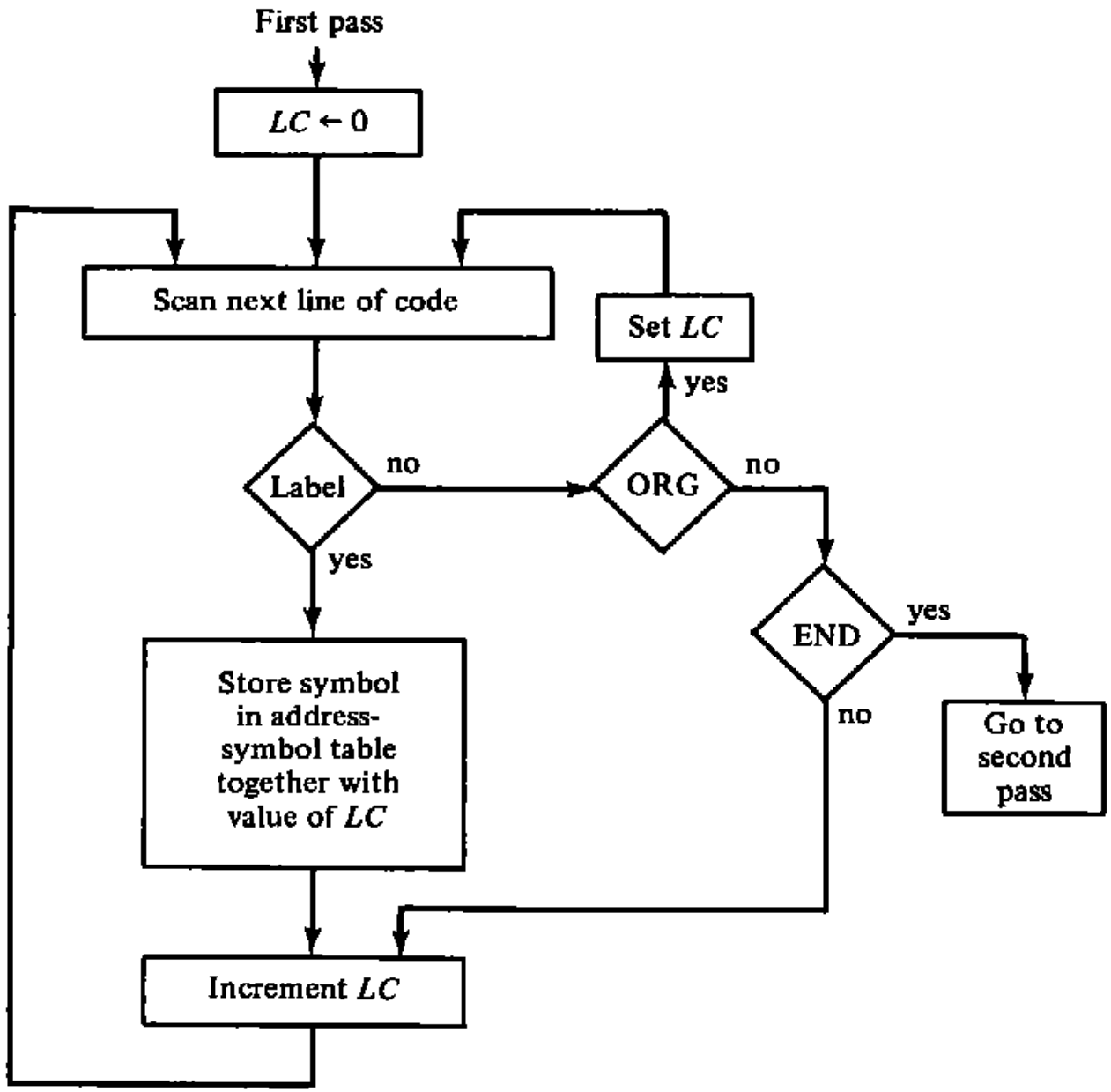
Symbol Table

Symbols	address
LOP	100
ONE	107
ZRO	10C
CTR	112
X	113
Y	114
P	115

How to do it automatically?

Assembler - first pass

Allocate addresses to all instructions and user defined symbols



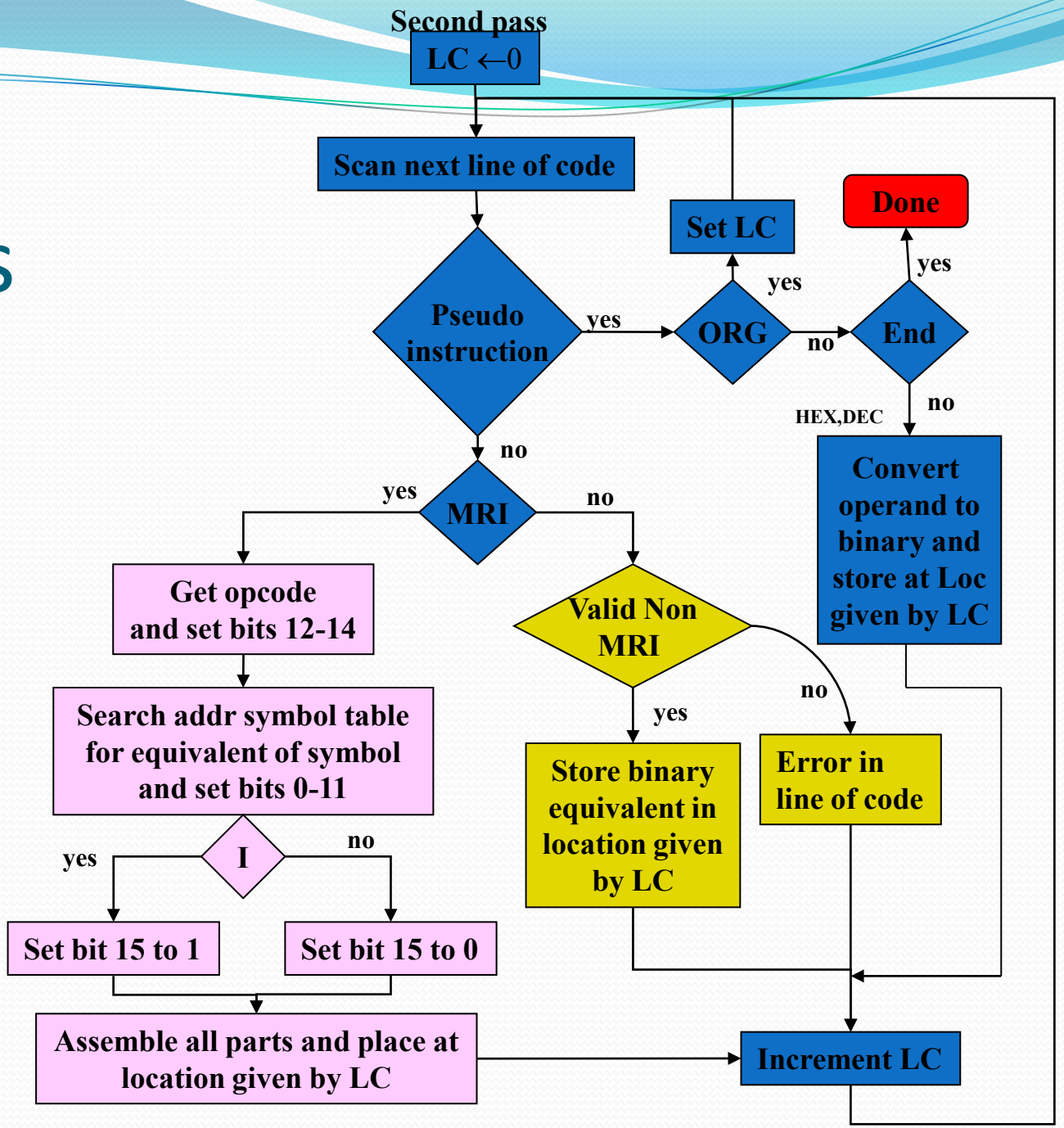
Assembler-second pass

A table look up procedure

- *Pseudo-instruction table*
- *MRI table*
- *Non-MRI table*
- *Address symbol table*

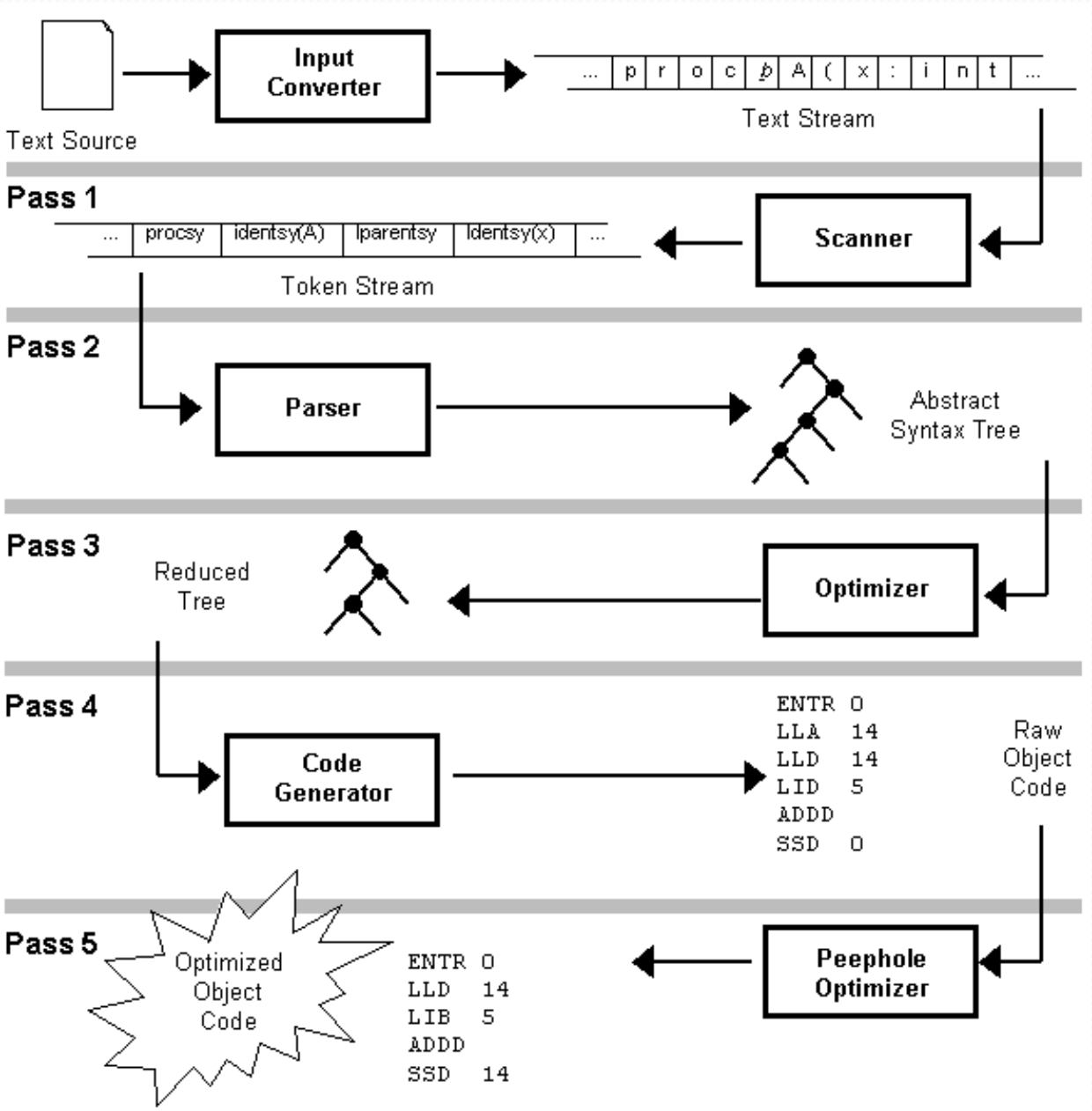
Want to add in more features?

Extend the flowchart…..



Compiler Basic

- It is not our job to write compilers.
- Among the 5 components, below 4 are standard. Many freeware with source code are available
  - Scanner
  - Parser
  - Optimizer
  - Peephole optimizer
- For a new ISA, only the module for code generator needs to be changed
  - no drastic difference in various ISA.
  - quick and dirty way is to define an equivalence of instruction for each instruction of an existing compiler



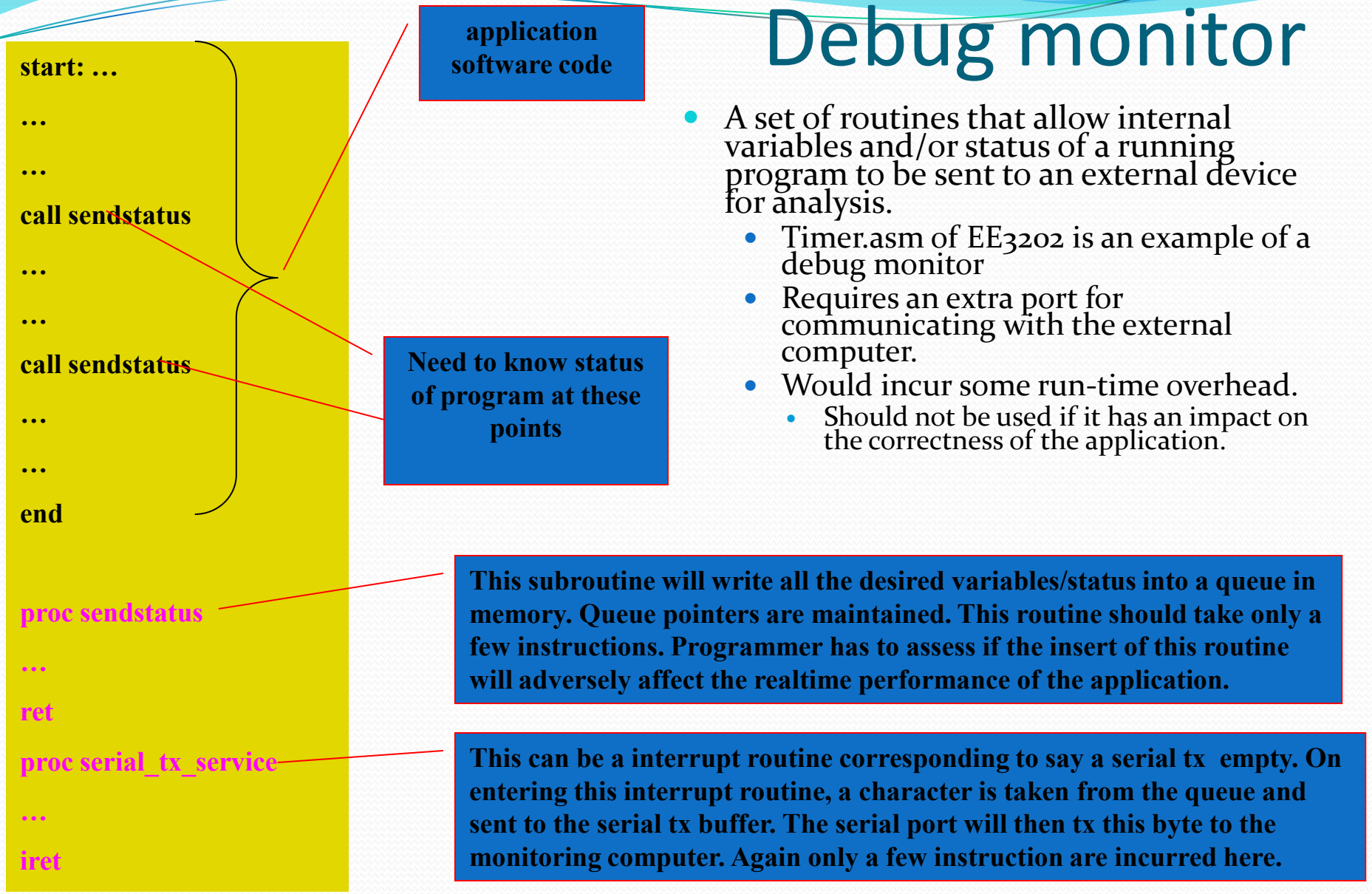
Code debugging

Simulator

Emulator

Debug monitor

- A set of routines that allow internal variables and/or status of a running program to be sent to an external device for analysis.
  - Timer.asm of EE3202 is an example of a debug monitor
  - Requires an extra port for communicating with the external computer.
  - Would incur some run-time overhead.
    - Should not be used if it has an impact on the correctness of the application.



Simulator

- Offline debugging of program
- A simulator is a program running on the PC that simulates the execution of the application program on a model of the application CPU.
  - Creates a copy each of all registers, flags and memory
  - Read application program, instruction at a time.
  - Perform operation and update registers, flags and memory in accordance to the intent of the instruction
  - Display user selectable parts of the registers/flag/memory on the screen
- Not real time, may in fact be very slow
  - many PC instructions needed to simulation one application CPU instruction
  - unable to simulate asynchronous triggering of events.
- Use mainly during initial testing of program

Emulator

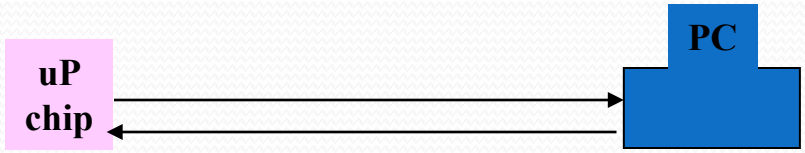
- Key functions to be provided by an emulator
  - Break points
    - When program reach a user specified location, the program is stopped.
    - Internal status of the program can be queried.
    - User can signal program to continue.
  - Single stepping
    - The program breaks after each instruction.
    - Internal status can be queried
    - User can signal program to execute next instruction
  - Program tracing – most important and useful
    - Provide a sequence of instructions executed by the processor from a user defined location.
  - Basic support infrastructure
    - An extra port for communicating with a PC
  - Breakpoint and single stepping can be implemented in software. Program tracing requires hardware support.

Overview: emulator

Hardware

Additional on-chip hardware

-Simple Serial Interface
-Interrupt based (lowest priority)
-Additional RAM – not user addressable
-Program trace logic



Software

uP software

Loader

Command interpreter

Breakpoint service routine

Single stepping service routine

Program trace service routine

PC software

Loader

Command translation routine

Result interpretation routine

Overview: emulator

- 2 operation modes
  - Normal: program is already debugged, uP is deployed
    - Upon reset, uP should run application program
    - Internal on-chip emulator (if still there in production lot) should lie dormant.
  - Debug: during development stage
    - Upon reset, uP should run internal on-chip emulator
    - User can download program,
    - Send commands to set breakpoint, program trace, single stepping
    - Send commands to upload results
  - Question:
    - How to ensure automatic performance of either mode?

Organization

Loc	Content of memory	Comments

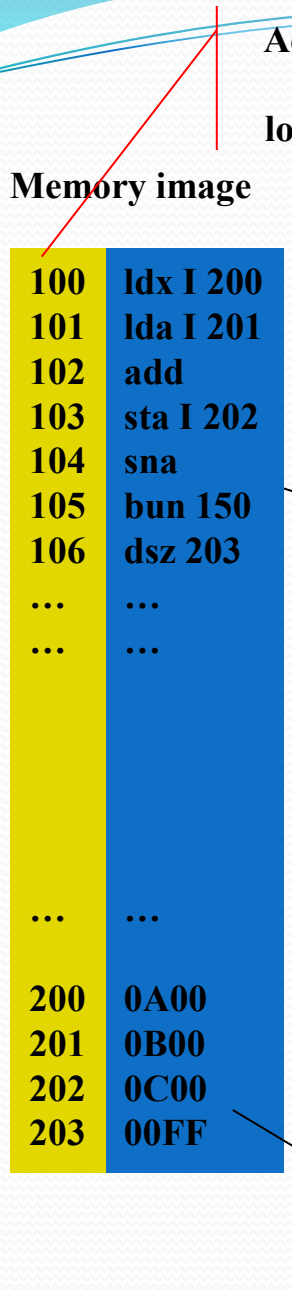
IRA: (Internal reset address)	Mov al, [testaddr]
	Cmp al,finalized
	Jne os_routine
	In al,serial port
	Cmp al, xx
	Jz os_routine
		IRA is the value initialized into the instruction pointer during reset. These 6 instructions are hard-coded into ROM. User have no access to these locations.

ERA: (External reset address)	User's application program …. …. …. …. …. ….	ERA is the reset address published in the uP manual. End user only knows of this address.
OSA: (OS address)	OS routines…… (including Loader Debugger)	

- On reset, all internal register are initialized.
- IP ← IRA
  - On recovery, instruction located at IRA is executed. Depending on the content of "testaddr" and the serial port buffer, either a jump to OS is executed or the program continues on to the ERA to execute the user application program.
- CPU to PC via the serial port.
  - A software program will run on the PC to manage the emulator section of the CPU.
- Content of "testaddr"
  - Debug mode (CPU is new), testaddr=oFFH.
    - Thus when ever the CPU is reset, the internal OS will be run
  - Normal mode, testaddr=ooH (finalised)
    - When the CPU is reset, the serial port will be read. If the serial buffer is "xx", then again the CPU jumps to the OS. Otherwise it will go on to the next location which is ERA.
    - If user wants to enter debug mode, it will have to send "xx" to the serial port and press reset at the same time.

Emulator: breakpoints

- Can be implemented in software



Mechanism

Job of the command interpreter to replace the location with the "call" upon receipt of the break address command from the PC

Suppose we want to break before sna is executed.

Before the program is run, save content of loc 104 somewhere and insert an instruction "call break_service" at that location.

Starts the program. When the program reaches loc 104, it will call break_service. "break_service" is a routine that allows internal parameters to be queried.

Emulator: single-stepping

Address of location

Memory image

100	ldx I 200
101	lda I 201
102	add
103	sta I 202
104	sna
105	bun 150
106	dsz 203
…	…
…	…
…	…
200	0A00
201	0B00
202	0C00
203	00FF

Content of memory

Mechanism

Command interpreter needs to perform this task

1. Command interpreter upon receiving "SS loc1", will save loc1 and instruction at loc1. It will then replace the content of loc1 with "bun SS_service".
2. The application program will be started.
3. When PC=loc1, bun SS_service will be executed.
4. SS_service is the beginning of a routine that will send the program status on serial port. It will then wait for user to send "continue".
5. Upon receipt of "continue,
   a. save content of loc1+1 and its content
   b. restore the content of loc1
   c. execute a bun loc1
6. The process repeats from Item (3) onwards.

Needs to write routine at this location

Program trace: operation manual

- Available resources – visible to user
  - PT register: holds address of instruction as program runs
  - En F/F: set to 1 to start trace from starting address
  - n words of RAM: hold addresses of program flow sequence
- Operation of trace
  - User loads starting address of program to trace from and starts program execution.
  - Command interpreter, on receipt,
    - load starting address into PT register
    - set EN f/f
    - Starts application program
  - As the program runs
    - No log takes place until PC equals PT
    - After PC goes pass PT, the first RAM location will be stored address of the first branch instruction that is taken.
      - Not taken branch will not be recorded
    - Address of all other instructions and conditional branch that are not taken will be ignored.
  - On completion
    - RAM contains the list of all the addresses of branch instructions that were taken as they occur.

Interpretation of logging convention

- Trace memory should not be big. Need to compress data
  - User has an image of program on the PC
  - User knows the starting point of the trace.
  - A long trace can be obtained with relatively small memory requirement.

Start here

```
100   LDA X
101   LDX Y
102   ADD
103   ADC
104   ADC
105   ADC
106   SCC          not taken
107   BUN 206
108   ADD
109   SCC          not taken
110   BUN 206      taken
...
206   SHR
207   SSC          taken
208   BUN 308
209   ADD
210   ADD
211   ADD
212   BUN 408      taken
```

Trace RAM
110
207
212

Only these 3 words will be sent to PC

On PC, we can expand into the following trace in consultation with the image of the program.

```
100
101
102
103
104
105
106
107
108
109
110   Since 110 contains BUN 206
206
207   Since 207 contains SCC
209
210
211
212   Since 212 contains BUN 408
408
...
```

Design of trace mechanism

- Available resources – visible to user
  - PT register: holds address of instruction as program runs
  - En F/F: set to 1 to start trace from starting address
  - n words of RAM: hold addresses of program flow sequence
- Additional resources
  - Tnew F/F: Tnew=1 indicates starting address not reached yet
  - A word comparator: To compare PC with PT. X=1 if PC=PT
  - SC1 F/F: State generator F/F. 2 states, T0 and T1
  - AR1 register Address register
- Design
  - (En)'T0: SC1←0
    - Before enable. Do nothing.
  - (En)(Tnew)(X)'(T0): SC1←0
    - Enabled, but PC has not reached starting address. Do nothing.
  - (En)(Tnew)(X)(T0): Tnew←0, M(AR1)←PC, PT←PT+1, SC1←0
    - Enabled and PC just reach starting address. Note that since AR1 is not increased, memory will continuously be overwritten.
  - (En)(Tnew)'(X)(T0): M(AR1)←PC, PT←PT+1, SC1←0
    - Enabled, tracing mode and no branch. Keep overwriting and increase PT to match PC's increment
  - (En)(Tnew)'(X)'(T0): AR1←AR1+1, PT←PC
  - (En)(Tnew)'(X)'(T1): PT←PT+1, SC1←0
    - Enabled, tracing mode and branch. Increase AR1, therefore preserving last stored address.
  - AR1=N: En←0, AR←0
    - No more trace memory.

Design of logic for program trace

(En)(Tnew)(X)(T0):	Tnew←0	CLR_{Tnew} =cond
(En)(Tnew)'(X)'(T0):	AR1←AR1+1	INR_{AR1}=cond
(En)(Tnew)'(X)'(T0):	PT←PC	LD_{PT}=cond
(En)(Tnew)'(X)'(T1):	PT←PT+1	INR_{PT}=OR(conditions)
(En)(Tnew)(X)(T0):	PT←PT+1	
(En)(Tnew)'(X)(T0):	PT←PT+1	
(En)(Tnew)(X)(T0):	M(AR1)←PC	WR_{MEM}=OR(conditions)
(En)(Tnew)'(X)(T0):	M(AR1)←PC	
(En)'T0:	SC1←0	CLR_{SC1}=OR(conditions)
(En)(Tnew)(X)'(T0):	SC1←0	
(En)(Tnew)'(X)(T0):	SC1←0	
(En)(Tnew)(X)(T0):	SC1←0	
(En)(Tnew)'(X)'(T1):	SC1←0	

Program Trace Circuit