Cooperative Threads for EdSim51

Pai H. Chou
National Tsing Hua University

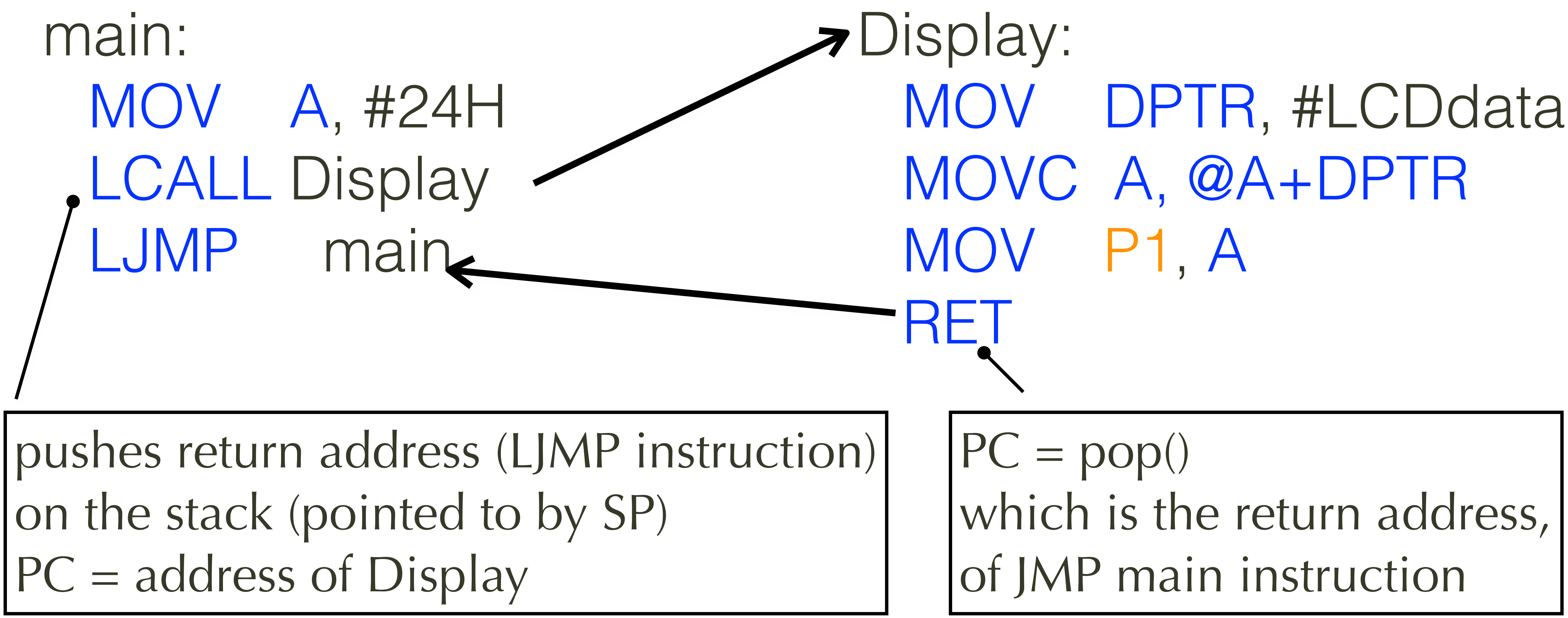
Outline

- Review
  - subroutine vs. function vs. thread
  - architecture review for EdSim51
- execution context on 8051
  - stack pointer, PSW
- register banking
- Threads API

Subroutine vs. Functions vs. Thread

- Subroutine (assembly)
    - **ISA-supported** call-return code fragment
    - assumes a (hardware-supported) stack, program counter
- Function (C)
    - **Compiler-supported** wrapper around subroutine
    - saving registers, passing parameters, auto-locals
    - assumes a stack, register set, program counter
- Thread (runtime support)
    - an execution context with its own copy of stack space, register values, and program counter
    - multiple threads run **concurrently** but share physical resources (I/O ports, global variables, etc)

Subroutine call



```
main:
  MOV    A, #24H
  LCALL Display
  LJMP     main
```

```
Display:
  MOV    DPTR, #LCDdata
  MOVC  A, @A+DPTR
  MOV    P1, A
  RET
```

pushes return address (LJMP instruction)
on the stack (pointed to by SP)
PC = address of Display

PC = pop()
which is the return address,
of JMP main instruction

Note: PC is <u>not</u> an SFRs; it is an internal register.
=> must use LCALL, LJMP, RET, or RETI to change it;
=> can't do a MOV instruction to/from PC.

Subroutine calls

- ISA-supported
  - LCALL, ACALL instructions => push(PC for next instruction), then jump
  - RET for return => PC = pop(return address) from stack
- Parameter passing
  - hardware does not dictate anything
  - could use accumulator (A), registers (R0-R7), stack, DPL/DPH, etc., depending on the compiler.
- Stack pointer (SP): SFR at 81H
  - power-up default at 07H (= empty stack; first item at 08H)
  - SP location can be saved and modified in assembly or in C
  - multi-byte order (e.g., code address): little-endian on 8051

Function Call

C code

```
void Main(void) {
   char i;
   for (i=0; i<10; i++) {
      DisplayLED(i);
   }
}
```

```
void DisplayLED(char c) {
   P1 = LED7seg(c);
}
```

```
char LED7seg(char c) {
   return LEDdata[c];
}
```

SDCC-Generated Assembly code

```
        mov     r7,#0x00
00102$:
        mov     dpl,r7
        push    ar7
        lcall   _DisplayLED
        pop     ar7
        inc     r7
        clr     c
        mov     a,r7
        xrl     a,#0x80
        subb    a,#0x8a
        jc      00102$
        ret
```

```
        lcall   _LED7seg
        mov     _P1,dpl
        ret
```

```
        mov     a,dpl
        mov     dptr,#_LED7seg_LEDdata_1_2
        movc    a,@a+dptr
        mov     dpl,a
        ret
```

Function calls

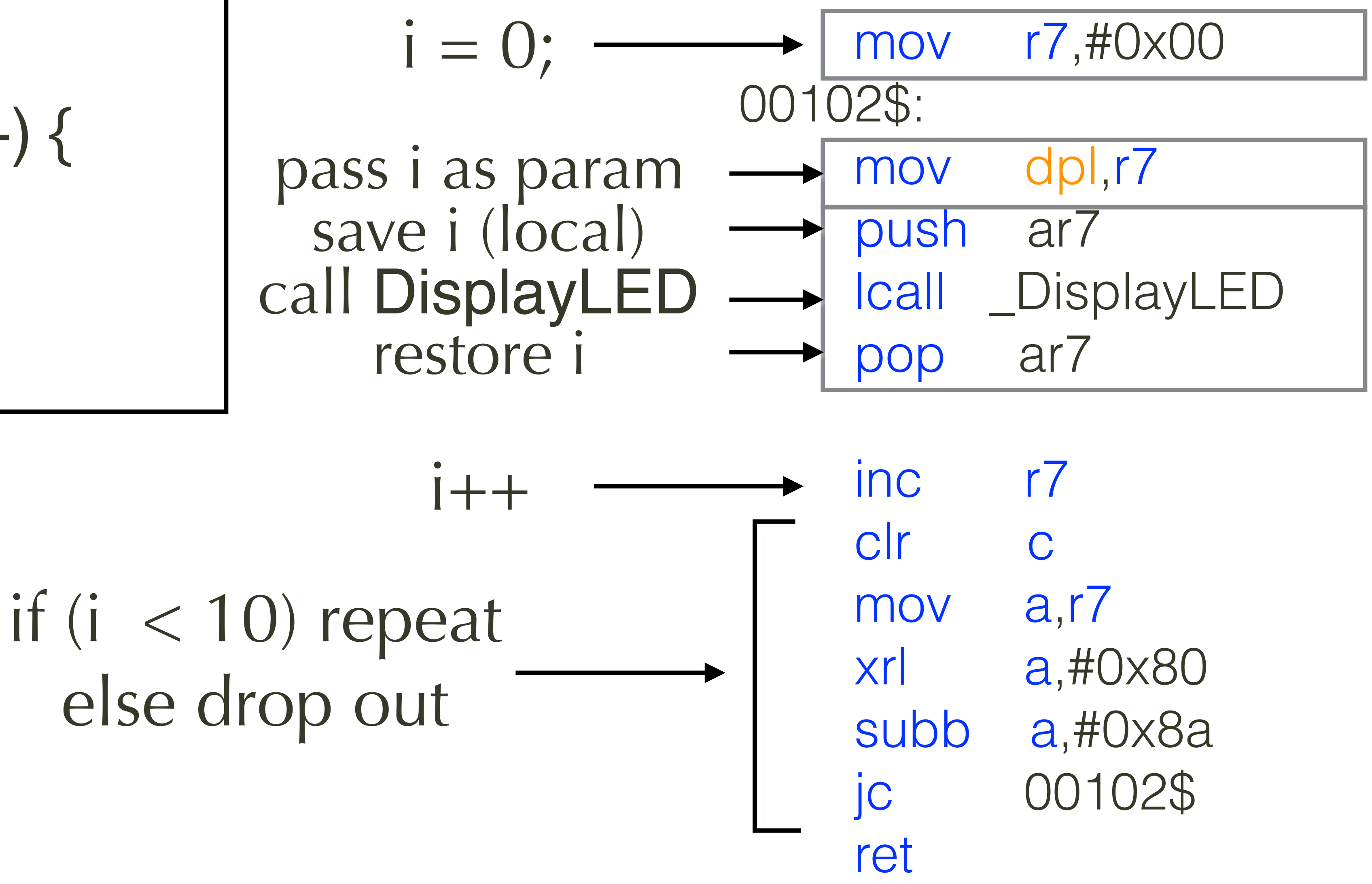
- Compiler-generated subroutine wrapper
  - essentially LCALL / RET, but plus other concepts
- Features
  - saving and restoring registers for caller/callee
  - define conventions for passing parameters
  - allocating and deallocating auto-local variables
  - handles return values from function call
- SDCC convention
  - uses DPL, DPH for passing parameters & return

Function Call: save & restore register

C code

```
void Main(void) {
   char i;
   for (i=0; i<10; i++) {
      DisplayLED(i);
   }
}
```

generated asm code



i = 0; → mov r7,#0x00

pass i as param → mov dpl,r7

save i (local) → push ar7

call DisplayLED → lcall _DisplayLED

restore i → pop ar7

i++ → inc r7

if (i < 10) repeat
else drop out → clr c / mov a,r7 / xrl a,#0x80 / subb a,#0x8a / jc 00102$

```
        mov     r7,#0x00
00102$:
        mov     dpl,r7
        push    ar7
        lcall   _DisplayLED
        pop     ar7
        inc     r7
        clr     c
        mov     a,r7
        xrl     a,#0x80
        subb    a,#0x8a
        jc      00102$
        ret
```

i: register r7
parameter: dpl

Function Call: return value

C code

```
void DisplayLED(char c) {
   P1 = LED7seg(c);
}
```

Assembly code

```
lcall   _LED7seg
mov     _P1,dpl
ret
```

(first) parameter char c is already in DPL register, and it is passed unchanged when calling LED7seg

return value from function call is also passed back in DPL

can you figure how how this function works?

```
char LED7seg(char c) {
   return LEDdata[c];
}
```

```
mov     a,dpl
mov     dptr,#_LED7seg_LEDdata_1_2
movc     a,@a+dptr
mov     dpl,a
ret
```

Threads

- Runtime support for multiple routines to run concurrently
- Each thread has its execution context
  - Its own registers & SFRs (R0-R7, ACC, B, DPTR, PSW)
  - Its own stack pointer (SP) value and stack space
  - Its own program counter (PC) value
- Threads shared resources
  - code, static globals, heap
  - => no protection from each other, but faster than process

Types of Threads

- Cooperative
  - A thread runs until it explicit **yield**s (or makes system call)
  - (+)Easy to implement, but (-) easy for a thread to hog CPU
- Preemptive
  - Timer interrupt preempts running thread, even if it does not yield
  - (-) Harder to implement, but (+) prevents CPU hogging
- Switching policy
  - various priority based selection of next thread to schedule
  - default: round-robin

Producer-Consumer Example

- Option 1: use three threads
  - `main()` is one thread, `Producer` & `Consumer` each gets a thread
  - easy, symmetric, but `main()`'s thread is wasted
- Option 2: use two threads
  - `main()` spawns `Consumer` thread; `main()` calls `Producer()`
  - more economical, reuse `main()`'s thread
- Shared-memory communication
  - 1-byte data buffer, 1-bit data-available flag
  - `Producer` thread-yields if buffer full (`Consumer` hasn't consumed it)
  - `Consumer` thread-yields
    - if buffer empty (i.e., `Producer` hasn't produced it yet)
    - if Tx busy (serial port hasn't finished writing it yet)

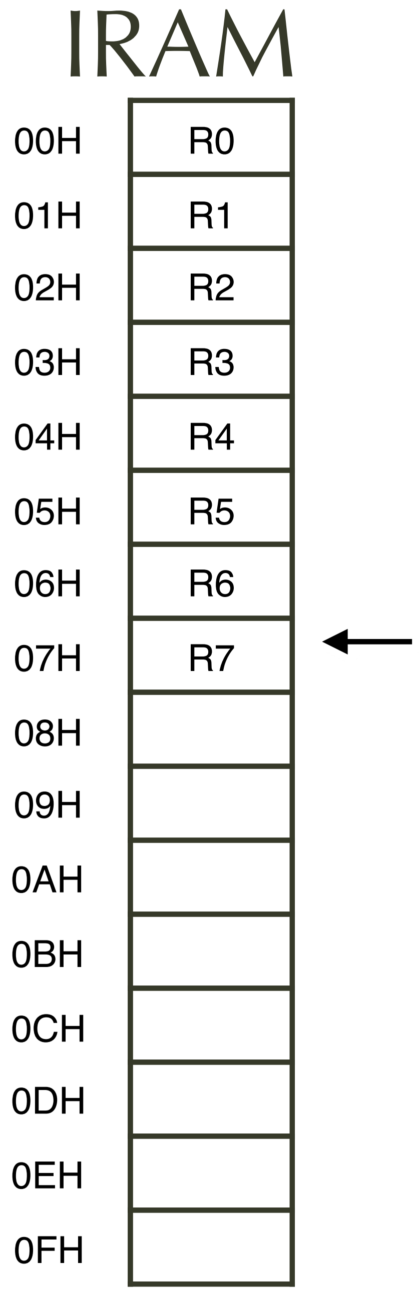
EdSim51 architecture review

- Harvard architecture
  - separate address spaces for code and data
- 64 KB code memory
  - => plenty!
- 128 Bytes of IRAM
  - address `00H` to `7FH`
- No XDATA on EdSim51!
  - even though 8051 can address up to 64 KB

IRAM usage

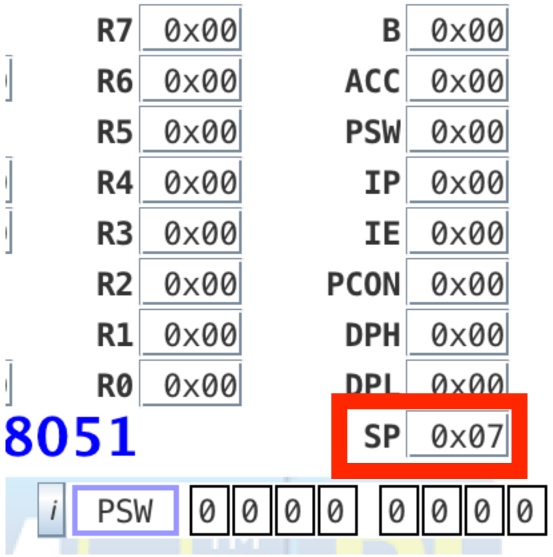
- Register banks: four sets of R0-R7 registers
  - bank 0 => address `00H` to `07H`
  - bank 1 => address `08H` to `0FH`,
  - bank 2 => address `10H` to `07H`
  - bank 3 => address `18H` to `1FH`
- Need to allocate the rest of IRAM for
  - scheduler's own storage, array of saved SP's (one per thread)
  - (per thread) stack for call/ret and saved ACC, B, DPTR, PSW

Hardware stack in 8051



IRAM

00H	R0
01H	R1
02H	R2
03H	R3
04H	R4
05H	R5
06H	R6
07H	R7 ←
08H	
09H	
0AH	
0BH	
0CH	
0DH	
0EH	
0FH	



R7	0x00	B	0x00
R6	0x00	ACC	0x00
R5	0x00	PSW	0x00
R4	0x00	IP	0x00
R3	0x00	IE	0x00
R2	0x00	PCON	0x00
R1	0x00	DPH	0x00
R0	0x00	DPL	0x00
8051		SP	0x07

PSW 0 0 0 0 0 0 0 0

- Power-up default
  - SP = 7
- PUSH
  - pre-increment then write
  - i.e., IRAM[++SP]=val
  - first push => address 8
- POP
  - read with post-decrement
  - i.e., return IRAM[SP--]
- User can assign SP to another value!

PSW: program status word

- SFR containing flags indicating status of 8051 CPU

CY (C bit)	PSW.7	Carry flag
AC	PSW.6	Auxiliary carry, for BCD arithmetic
F0, --	PSW.5, .1	(user)
RS1	PSW.4	**Register bank select**
RS0	PSW.3	
OV	PSW.2	Overflow
P	PSW.0	Parity: even or odd# of 1's in A

Data structures needed (instead of a threads control block)

- For each thread
  - stack space (for execution and saving PC,A,B,DPTR,PSW)
  - register bank (for saving R0-R7)
  - saved stack pointer (SP) - entry in savedSP array
- For threads manager
  - bitmap for which thread is active, optionally #threads
  - current thread ID
  - other temporary variables

source files

- `testcoop.c`
  - startup code, main program, producer, consumer, shared variable
- `coopertive.c`
  - bootstrapping code called by startup code,
  - thread creation, yield, exit
- `cooperative.h`
  - API to be called by `testcoop.c`

File: cooperative.h

- #define MAXTHREADS 4
- typedef char ThreadID; // single-byte ID for threads 0..3; -1 => invalid.
- typedef void (*FunctionPtr)(void); // 2-byte (code-space) pointer to function
- ThreadID ThreadCreate(FunctionPtr fp)
    - create and start a thread to run function fp
- void ThreadYield(void)
    - current thread switches itself out, lets another thread run
    - => includes picking the next available thread in round-robin.
      => later we may want to support different policies.
- void ThreadExit(void)
    - called by the current thread to terminate itself

File: testcoop.c

-
  ```
  #include <8051.h>
  #include "cooperative.h"
  ```

-
  ```
  __data __at (address) var...;  // declare global variables for shared var
  ```

-
  ```
  void Producer(void) { .. }
  void Consumer(void) { .. }
  void main(void) { .. }
  ```

-
  ```
  void _sdcc_gsinit_startup(void) {
          __asm
                  ljmp  _Bootstrap
          __endasm;
  }
  ```

-
  ```
  void _mcs51_genRAMCLEAR(void) {}
  void _mcs51_genXINIT(void) {}
  void _mcs51_genXRAMCLEAR(void) {}
  ```

testcoop.c: Producer/Consumer

- Similar idea to the python code last week
- Producer: loop forever
  - poll buffer, full => ThreadYield();
  - buffer available => make next item, mark buffer full
- Consumer: initialize UART Tx, then loop forever
  - poll buffer, empty => ThreadYield();
  - buffer available => read it, write to UART Tx
  - poll UART Tx, busy => ThreadYield();

testcoop.c: main()

- runs in its own thread
  - created by Bootstrap code upon startup
- Need two threads
  - create one thread for producer or consumer
  - either create another thread or reuse its own thread to run the other (consumer or producer)
- Does not return

File: cooperative.c

- #include <8051.h>
  #include "cooperative.h"
- variables for the thread package
- macros for SAVESTATE and RESTORESTATE
- extern void main(void);
- void Bootstrap(void) { .. }
- ThreadID ThreadCreate(FunctionPtr) { .. }
- void ThreadYield(void) { ... }
- void ThreadExit(void) { ... }

Memory allocation

- For each thread

	thread 0	thread 1	thread 2	thread 3
stack	40H-4FH	50H-5FH	60H-6FH	70H-7FH
reg bank	00H-07H	08H-0FH	10H-17F	18H-1FH
saved SP	30H?	31H?	32H?	33H?

- For threads package

purpose	**address**
bitmap for active threads	34H?
current thread's ID	35H?
other temps	36H?

- For producer/consumer/main etc

cooperative.c: internal code

- SAVESTATE
  - push ACC, B, DPTR, PSW onto stack
    as part of pushing PSW, also saves register bank
  - save stack pointer for the current thread
  - Defined as C macros written in inlined assembly
- RESTORESTATE
  - reverse operation of SAVESTATE
- void Bootstrap(void)
  - start-up code to set up and run the first thread

Illustration of context switching: Powering up

addr	use	_0H-_7H	_8H-_FH
0_H	bank 0 - 1	bank 0 active	initial stack
1_H	bank 2 -3	-	-
2_H	globals	bit & byte addressable	
3_H	globals	byte addressable	
4_H	stack 0	-	-
5_H	stack 1	-	-
6_H	stack 2	-	-
7_H	stack 3	-	-

hw reg	value
SP	07H
PC	0000H
PSW	00H

globals	value
thread bitmap	-
current thread	-
savedSP[0:3]	-, -, -, -

```
void _sdcc_gsinit_startup(void) {
  __asm
    ljmp _Bootstrap
  __endasm;
}
```

Illustration: Bootstrap (1)

addr	use	_0H-_7H	_8H-_FH
0_H	bank 0 - 1	bank 0 active	initial stack
1_H	bank 2 -3	-	-
2_H	globals	bit & byte addressable	
3_H	globals	byte addressable	
4_H	stack 0	-	-
5_H	stack 1	-	-
6_H	stack 2	-	-
7_H	stack 3	-	-

hw reg	value
SP	07H
PC	Bootstrap(1)
PSW	00H

globals	value
thread bitmap	**0000B**
current thread	-
savedSP[0:3]	-, -, -, -

```
void Bootstrap(void) {
  // (1) initialize thread mgr vars
  // (2) create thread for main
  // (3) set current thread ID
  // (4) restore
}
```

Illustration: Bootstrap (2) on calling ThreadCreate(main)

addr	use	_0H-_7H	_8H-_FH
0_H	bank 0 - 1	bank 0 active	initial stack
1_H	bank 2 -3	-	-
2_H	globals	bit & byte addressable	
3_H	globals	byte addressable	
4_H	stack 0	-	-
5_H	stack 1	-	-
6_H	stack 2	-	-
7_H	stack 3	-	-

initial stack (assuming initialized to 07H)

08H	return addr = Bootstrap (3)
09H	
0AH	
0BH	
0CH	
..	...

hw reg	value
SP	09H
PC	ThreadCreate
PSW	00H
DPL	address of main
DPH	

```
void Bootstrap(void) {
  // (1) initialize thread mgr vars
  // (2) create thread for main
  // (3) set current thread ID
  // (4) restore
}
```

Illustration: Bootstrap(2) on returning from ThreadCreate

addr	use	_0H-_7H	_8H-_FH
0_H	bank 0 - 1	bank 0 active	initial stack
1_H	bank 2 -3	-	-
2_H	globals	bit & byte addressable	
3_H	globals	byte addressable	
4_H	stack 0	stack for thread 0	
5_H	stack 1	-	-
6_H	stack 2	-	-
7_H	stack 3	-	-

40H	main's address
41H	
42H	0H for ACC
43H	0H for B
44H	0H for DPL
45H	0H for DPH
46H	0H for PSW
...	
4FH	

hw reg	value
SP	07H
PC	Bootstrap(3)
DPL	threadID = 0

```
void Bootstrap(void) {
  // (1) initialize thread mgr vars
  // (2) create thread for main
  // (3) set current thread ID
  // (4) restore
}
```

globals	value
thread bitmap	**0001B**
current thread	-
savedSP[0:3]	**46H**, -, -, -

Illustration: Bootstrap(3)

addr	use	_0H-_7H	_8H-_FH
0_H	bank 0 - 1	bank 0 active	initial stack
1_H	bank 2 -3	-	-
2_H	globals	bit & byte addressable	
3_H	globals	byte addressable	
4_H	stack 0	stack for thread 0	
5_H	stack 1	-	-
6_H	stack 2	-	-
7_H	stack 3	-	-

40H	main's address
41H	
42H	ACC (th0)
43H	B (th0)
44H	DPL (th0)
45H	DPH (th0)
46H	PSW (th0)
47H	
48H	
...	
4FH	

```
void Bootstrap(void) {
  // (1) initialize thread mgr vars
  // (2) create thread for main
  // (3) set current thread ID
  // (4) restore
}
```

purpose	value
thread bitmap	0001B
current thread	**0**
savedSP[0:3]	46H, -, -, -

Illustration: Bootstrap (4)

addr	use	_0H-_7H	_8H-_FH
0_H	bank 0 - 1	bank 0 active	-
1_H	bank 2 -3	-	-
2_H	globals	bit & byte addressable	
3_H	globals	byte addressable	
4_H	stack 0	stack for thread 0	
5_H	stack 1	-	-
6_H	stack 2	-	-
7_H	stack 3	-	-

SP	**46H => 3FH**
PC	Bootstrap(4) => main (stack 41H-40H)
PSW	from stack 46H
DPTR	from stack 45-44
B	from stack 43H
ACC	from stack 42H

```
void Bootstrap(void) {
  // (1) initialize thread mgr vars
  // (2) create thread for main
  // (3) set current thread ID
  // (4) restore
}
```

On finishing Bootstrap

- stack was set up by ThreadCreate(main)
  - RESTORESTATE sets SP to the savedSP for stack-0, restores its PSW (which selects register bank 0), DPTR, B, ACC using stack value
  - stack 0 now has the return address of main
- Bootstrap does a RET to main()
  - PC is now pointing to main
  - stack 0 is now empty: SP == 0x3F
    => this means main() should not return!!
    (option: could set up each thread's stack by pushing ThreadExit's address first)

Illustration: main (1) on calling ThreadCreate(Producer)

addr	use	_0H-_7H	_8H-_FH
0_H	bank 0 - 1	bank 0 active	-
1_H	bank 2 -3	-	-
2_H	globals	bit & byte addressable	
3_H	globals	byte addressable	
4_H	stack 0	stack for thread 0	
5_H	stack 1	-	-
6_H	stack 2	-	-
7_H	stack 3	-	-

40H	return address = main (2)
41H	
42H	
43H	
44H	
..	...

SP	41H (stack 0)
PC	ThreadCreate
PSW	00H
DPL	address of Producer
DPH	

```
void main(void) {
  // (1) create thread for Producer
  // (2) call Consumer
}
```

Illustration: main(1) on returning from ThreadCreate(Producer)

addr	use	_0H-_7H	_8H-_FH
0_H	bank 0 - 1	bank 0 active	bank 1
1_H	bank 2 -3	-	-
2_H	globals	bit & byte addressable	
3_H	globals	byte addressable	
4_H	stack 0	stack for thread 0	
5_H	stack 1	stack for thread 1	
6_H	stack 2	-	-
7_H	stack 3	-	-

50H	Producer's address
51H	
52H	ACC=0
53H	B=0
54H	DPL=0
55H	DPH=0
56H	PSW=**08H**
...	
5FH	

SP	3FH (stack 0)
PC	main(2)
DPL	threadID = 1

```
void main(void) {
  // (1) create thread for Producer
  // (2) call Consumer
}
```

globals	value
thread bitmap	**0011B**
current thread	0
savedSP[0:3]	46H, **56H**, -, -

Illustration: main(2) calling Consumer

addr	use	_0H-_7H	_8H-_FH
0_H	bank 0 - 1	bank 0 active	bank 1
1_H	bank 2 -3	-	-
2_H	globals	bit & byte addressable	
3_H	globals	byte addressable	
4_H	stack 0	stack for thread 0	
5_H	stack 1	stack for thread 1	
6_H	stack 2	-	-
7_H	stack 3	-	-

40H	return address = main (3)
41H	
42H	
43H	
44H	
..	...

SP	41H (stack 0)
PC	Consumer
PSW	-
DPL	-
DPH	

```
void main(void) {
  // (1) create thread for Producer
  // (2) call Consumer
  // (3)
}
```

Illustration: Consumer yields

addr	use	_0H-_7H	_8H-_FH
0_H	bank 0 - 1	bank 0 active	bank 1
1_H	bank 2 -3	-	-
2_H	globals	bit & byte addressable	
3_H	globals	byte addressable	
4_H	stack 0	stack for thread 0	
5_H	stack 1	stack for thread 1	
6_H	stack 2	-	-
7_H	stack 3	-	-

40H	return address = main (3)
41H	
42H	return address = Consumer (2)
43H	
44H	
..	...

SP	43H (stack 0)
PC	ThreadYield
PSW	-
DPL	-
DPH	

```
void Consumer(void) {
  // (1) poll buffer, busy => ThreadYield()
  // (2)
  // (3)
}
```

Illustration: ThreadYield SAVESTATE

addr	use	_0H-_7H	_8H-_FH
0_H	**bank 0 - 1**	bank 0 active	bank 1
1_H	**bank 2 -3**	-	-
2_H	**globals**	bit & byte addressable	
3_H	**globals**	byte addressable	
4_H	**stack 0**	stack for thread 0	
5_H	**stack 1**	stack for thread 1	
6_H	**stack 2**	-	-
7_H	**stack 3**	-	-

40H	address of main(3)
41H	
42H	address of Consumer(2)
43H	
44H	push ACC
45H	push B
46H	push DPL
47H	push DPH
48H	push PSW
49H	
...	

```
void ThreadYield(void) {
  // (1) SAVESTATE
  // (2) pick next thread
  // (3) RESTORESTATE
}
```

globals	**value**
thread bitmap	0011B
current thread	0
savedSP[0:3]	**48H**, 56H, 00, 00

Illustration: ThreadYield picking next thread

addr	use	_0H-_7H	_8H-_FH
0_H	**bank 0 - 1**	bank 0 active	bank 1
1_H	**bank 2 -3**	-	-
2_H	**globals**	bit & byte addressable	
3_H	**globals**	byte addressable	
4_H	**stack 0**	stack for thread 0	
5_H	**stack 1**	stack for thread 1	
6_H	**stack 2**	-	-
7_H	**stack 3**	-	-

40H	address of main(3)
41H	
42H	address of Consumer(2)
43H	
44H	ACC (th0)
45H	B (th0)
46H	DPL (th0)
47H	DPH (th0)
48H	PSW (th0)
49H	
...	

```
void ThreadYield(void) {
  // (1) SAVESTATE
  // (2) pick next thread
  // (3) RESTORESTATE
}
```

globals	value
thread bitmap	0011B
current thread	**0 => 1**
savedSP[0:3]	48H, 56H, -, -

Illustration: ThreadYield

RESTORESTATE

addr	use	_0H-_7H	_8H-_FH
0_H	bank 0 - 1	bank 0	bank 1 active
1_H	bank 2 -3	-	-
2_H	globals	bit & byte addressable	
3_H	globals	byte addressable	
4_H	stack 0	stack for thread 0	
5_H	stack 1	stack for thread 1	
6_H	stack 2	-	-
7_H	stack 3	-	-

50H	address of Producer
51H	
52H	pop ACC
53H	pop B
54H	pop DPL
55H	pop DPH
56H	pop PSW
…	
5FH	

SP	**56H => 51H**
PC	ThreadYield(3) => Producer (stack 51H-50H)
PSW	from stack 56H
DPTR	from stack 55-54
B	from stack 53H
ACC	from stack 52H

```
void ThreadYield(void) {
  // (1) SAVESTATE
  // (2) pick next thread
  // (3) RESTORESTATE
}
```

globals	value
thread bitmap	0011B
current thread	1
savedSP[0:3]	48H, 56H,

On finishing ThreadYield

- stack was set up by ThreadCreate(Producer)
    - RESTORESTATE sets SP to stack 1,
      restores its PSW (0x08 selects register bank 1), DPTR, B, ACC using stack value
    - stack 1 now has the return address of Producer
- ThreadYield() does a RET to Producer()
    - PC is now pointing to Producer
    - stack 1 is now empty: SP == 0x4F
      => in this version, Producer() should never return as implicit ThreadExit(), because stack would underflow.

Summary: Cooperative Threads

- ThreadYield()
  - saves context of current thread
  - selects next thread (could be same if only one)
  - restore context of (new) current thread
- Context
  - stack contains return address for resuming thread
  - each thread's stack contains register values and bank info; register banking enables quick switch
  - ultimately, each thread's stack pointer is handle to everything

Issues with Current Version of Cooperative Multithreading

- Two ways to call ThreadExit()
  - Explicitly calling ThreadExit()
  - Implicitly: ThreadCreate() needs to push ThreadExit()'s address to bottom of each thread's stack => extra space, but may be safer
- State of thread
  - currently assumes Ready; may need to add Waiting
- Open issues:
  - Scheduling Policy: defaults round-robin policy; need priority
  - Preemption: need atomic operations and timer interrupt
  - Requiring ThreadJoin() or ThreadDetach()?