**Assembly coding ABC-1**

1. phenomena behind statement execution in a program

|  |  |
| --- | --- |
| statements | behind the scene |
| int x, y, z; // (0)  . . .  x= 1; // (1)  y= 2; // (2)  z= x + y; // (3)  . . . | context in which a program is running:  CPU  **PC** 00000000H:  decoder 32-bit 。。。  addr-gen **addr. bus**  R0 ADDRx:  。。。 32-bit ADDRy:  Rn **data bus** ADDRz:  。。。    ffffffffH:   1. 4-byte space reserved for each of the int. variables,   starting respectively at address at ADDRx, ADDRy,  and ADDRz   1. transfer of a 32-bit value of 1 to memory location   ADDRx  CPU  **PC** 00000000H:  decoder **ADDRx** 。。。  addr-gen  。。。 ADDRx:  Rj: 1 **00000001H**  。。。  :  。。。    ffffffffH:   1. transfer of a 32-bit value of 2 to memory location   ADDRy |

|  |  |
| --- | --- |
| statements | behind the scene |
| int x, y, z; // (0)  . . .  x= 1; // (1)  y= 2; // (2)  z= x + y; // (3)  . . . | CPU  **PC** 00000000H:  decoder **ADDRy** 。。。  addr-gen  。。。  Rk: 2 **00000002H** ADDRy:  。。。  :  。。。    ffffffffH:  (3) **z = x + y;**   * 1. transfer of a 32-bit value of 1 from memory location ADDRx to an internal register   MOV REGi, ADDRx  CPU  **PC** 00000000H:  decoder **ADDRx** 。。。  addr-gen  。。。 ADDRx:  Ri **00000001H**  。。。  :  。。。    ffffffffH:   * 1. transfer of a 32-bit value of 2 from memory location ADDRy to an interna register   MOV REGi, ADDRy |

|  |  |
| --- | --- |
| statements | behind the scene |
| int x, y, z; // (0)  . . .  x= 1; // (1)  y= 2; // (2)  z= x + y; // (3)  . . . | CPU  **PC** 00000000H:  decoder **ADDRy** 。。。  addr-gen  。。。  Rj **00000002H** ADDRy:  。。。  :  。。。    ffffffffH:   * 1. adding of two registers, sum saved in a register   ADD REGk, REGi, REGj  CPU  **PC** 00000000H:  decoder xxxxxxxxH 。。。  addr-gen  。。。 ADDRx:  Rk **xxxxxxxxH**  。。。  :  Ri Rj  。。。    ffffffffH: |

|  |  |
| --- | --- |
| statements | behind the scene |
| int x, y, z; // (0)  . . .  x= 1; // (1)  y= 2; // (2)  z= x + y; // (3)  . . . | * 1. transfer of a 32-bit value of 3 to memory location ADDRz   MOV ADDRx, REGk  CPU  **PC** 00000000H:  decoder **ADDRz** 。。。  addr-gen  。。。  Rk:3 **00000003H**  。。。  ADDRz:  。。。    ffffffffH: |
| z = x + y;  code compilation  MOV REGi, ADDRx  MOV REGj, ADDRy  ADD REGk, REGi, REGj  MOV ADDRx, REGk  assembler  machine codes | |

1. a pseudo-CPU TWN#1 and some examples to start with

|  |  |
| --- | --- |
| . . .  // moving a block of 8 bytes // data starting at 1000H  // to the block at 2000H  mov 2000H, 1000H  mov 2001H, 1001H  mov 2002H, 1002H  mov 2003H, 1003H  mov 2004H, 1004H  mov 2005H, 1005H  mov 2006H, 1006H  mov 2007H, 1007H  . . .  //what if moving 256 bytes //at 1000H. . .  - - - - - - - - - - - - - - - - - - - - - -  // adding up 8 consecutive  //bytes of data starting at  //1000H; with the sum  //saved in R0  . . .  ADD R0, 1001H, 1002H  ADD R0, 1003H, R0  ADD R0, 1004H, R0  ADD R0, 1005H, R0  ADD R0, 1006H, R0  ADD R0, 1007H, R0  . . .  // again, what if . . . | Given a pseudo-CPU “TWN#1”  \* with 64KB memory data space (16 bits of address, that is): 0000H ~ FFFFH  \* 16 8-bit registers: R0~R15 |
| computing capacity of TWN#1  \* data transfer in byte – m2m, r2r, r2m, m2r  MOV destination, source  \* arithmetic-ops on byte – m&m, r&r, m&r, r&m  ADD sum, addend, adder  SUB difference, minuend, subtrahend  MUL product, multiplicand, multiplier  DIV quotient, remainder, dividend, divisor  INC opnd1  DEC opnd1  \* logic-ops on byte – m&m, r&r, m&r, r&m  AND opnd1, opnd2, opnd3  OR opnd1, opnd2, opnd3  XOR opnd1, opnd2, opnd3  CLR opnd1  NOT opnd1  RR opnd1  RL opnd2  \* no support of memory-indexing access  (i.e., memory units can not be accessed via index/pointer, but by address only)  \* no support of “initial value setting”  (i.e. mov R0, “999” not available  For initializing R0 with 999) |

|  |  |
| --- | --- |
| //adding data byte in block1 //with corresponding data //byte in block2, saving the  //sum in block3  . . .  ADD 3000H, 2000H, 1000H  ADD 3001H, 2001H, 1001H  ADD 3002H, 2002H, 1002H  ADD 3003H, 2003H, 1003H  ADD 3004H, 2004H, 1004H  ADD 3005H, 2005H, 1005H  ADD 3006H, 2006H, 1006H  ADD 3007H, 2007H, 1007H  . . .  //again, what if . . .  - - - - - - - - - - - - - - - - - - - - - -  // guess what the code piece // below is doing.  . . .  CLR R0  INC R0  RL R0  INC R0  RL R0  RL R0  RL R0  INC R0  RL R0  INC R0  . . .  // can you try writing a  // functionally equivalent  //code piece? Any variants  // to write? | Given a pseudo-CPU “TWN#1”  \* with 64KB memory data space (16 bits of address, that is): 0000H ~ FFFFH  \* 16 8-bit registers: R0~R15 |
| computing capacity of TWN#1  \* data transfer in byte – m2m, r2r, r2m, m2r  MOV destination, source  \* arithmetic-ops on byte – m&m, r&r, m&r, r&m  ADD sum, addend, adder  SUB difference, minuend, subtrahend  MUL product, multiplicand, multiplier  DIV quotient, remainder, dividend, divisor  INC opnd1  DEC opnd1  \* logic-ops on byte – m&m, r&r, m&r, r&m  AND opnd1, opnd2, opnd3  OR opnd1, opnd2, opnd3  XOR opnd1, opnd2, opnd3  CLR opnd1  NOT opnd1  RR opnd1  RL opnd2  \* no support of memory-indexing access  (i.e., memory units cannot be accessed via index/pointer, but by address only)  \* no support of “initial value setting”  (i.e. mov R0, “999” not available  For initializing R0 with 999) |

|  |  |
| --- | --- |
| // and what is this for?  . . .  NEG R1  INC R1  ADD R2, R1, R2  . . .  // any alternative coding to  // the computation above?  - - - - - - - - - - - - - - - - - - - - - -  // multiplication  . . .  MUL 1000H, R0, R1  . . .  // what concern you might  // have if told that the 16-bit // product is saved in 2 bytes // starting at 1000H?  - - - - - - - - - - - - - - - - - - - - - -  // what to do with the //addition of two 16-bit  //unsigned integers?  // \*data setting in memory?  // \* data setting in registers?  // \* is ADD still applicable?  // \* other concerns? | Given a pseudo-CPU “TWN#1”  \* with 64KB memory data space (16 bits of address, that is): 0000H ~ FFFFH  \* 16 8-bit registers: R0~R15 |
| computing capacity of TWN#1  \* data transfer in byte – m2m, r2r, r2m, m2r  MOV destination, source  \* arithmetic-ops on byte – m&m, r&r, m&r,  r&m  ADD sum, addend, adder  SUB difference, minuend, subtrahend  MUL product, multiplicand, multiplier  DIV quotient, remainder, dividend, divisor  INC opnd1  DEC opnd1  \* logic-ops on byte – m&m, r&r, m&r, r&m  AND opnd1, opnd2, opnd3  OR opnd1, opnd2, opnd3  XOR opnd1, opnd2, opnd3  CLR opnd1  NOT opnd1  RR opnd1  RL opnd2  \* no support of memory-indexing access  (i.e., memory units cannot be accessed via index/pointer, but by address only)  \* no support of “initial value setting”  (i.e. mov R0, “999” not available  For initializing R0 with 999) |

1. Pseudo-CPU TWN#2 and data representation issues

|  |  |
| --- | --- |
| // block-data transfer  . . .  mov R7, #255  movdp DPTR1, #1000H  movdp DPTR2, #2000H  next:  mov R0, DPTR1  mov DPTR2, R0  inc DPTR1  inc DPTR2  DEC R7  JR7NZ next  . . .  // the code above shows a  // typical loo-structure . . .  - - - - - - - - - - - - - - - - - - - - - -  // suppose a block of 1000  // data bytes are to be moved  // to a new memory location  // . . .  // coding 5 loops, each  // taking care of 200 bytes  // transfer, would suffice  // . . .  // alternative:  // a 2-layered looping  // (nested-loop of depth  // two) works as well;  // and HOW?  - - - - - - - - - - - - - - - - - - - - - -  // what about moving 10  //blocks of data, 255-byte in  //length, respectively to 10  //new memory locations? | Given the pseudo-CPU “TWN#1”  \* with 64KB memory data space (16 bits of address, that is): 0000H ~ FFFFH  \* 16 8-bit registers: R0~R15  \* computing capacity with the original Inst.-set |
| *any missing elements in TWN#1 architecture :*  *- status/context tracking*  *- data representation*  *- Inst.-set enhancement*  *if serious data-manipulation is attempted?* |
| *//////////////////////////////////////////////////* |
| upgraded pseudo-CPU “**TWN#2**”  \* with 64KB memory data space (16 bits of address, that is): 0000H ~ FFFFH  \* 16 8-bit registers: R0~R15  \* CPU status register   |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | Cy | Z | P |  |  |  |  |  |   \* 16-bit data-pointer: DPTR1, DPTR2  \* 16-bit stack pointer: SP  \* memory-data residence in little-endian order  \* |
| computing capacity  \* support of 8-bit BYTE operations  - 8 individual bits in logic operations  - unsigned 8-bit data in arithmetic operations  \* support of 16-bit memory-indexing for  - data access  - STACK operation  \* Inst. –set enhancement  - initial-value setting  - indexing mechanism for memory access  - STACK operation  - non-sequential code execution control  - data comparison  - . . . |

|  |  |
| --- | --- |
| // what is the following code  // for?  . . .  mov SP, #1000H  push 400H  push 401H  push 402H  push 403H  push 404H  push 405H  push 406H  push 407H  pop 500H  pop 501H  pop 502H  pop 503H  pop 504H  pop 505H  pop 506H  pop 507H  . . .  //status of TWN#2  // architecture?  - - - - - - - - - - - - - - - - - - - - - -  // consider two blocks, each  //with 255 16-bit data units  //stored in little-endian  //format; try writing a code  //piece to adding two corres-  //ponding data units in the 2  //blocks, save the sum in a  //3rd block  - - - - - - - - - - - - - - - - - - - - - - | computing capacity ( TWN#2 continued)  **- MOV opnd1, DPTRi**  **MOV DPTRi, opnd2**  **INC DPTRi**  **DEC DPTRi**  ; memory data item indexed by the address  ; held in DPTR is accessed (verbally: the data  ; pointed to by DPTR)  ; indexing operation for a data-item in  ; memory  ; opnd1 🡨 (DPTRi)  ; (DPTRi) 🡨 opnd2  **- MOVDP DPTRi, m\_opnd**  **MOVDP m\_opnd, DPTRi**  **MOVDP DPTRi, #nnnn**  ; DPTRi\_LOW  **- PUSH opnd1**  ; opnd1 is “PUSHed” on top of the STACK  ; as designated/indexed by SP  ;  **- POP opnd1**  ; data item on top of the STACK, as indexed  ; by SP, is “POPed” off and saved in opnd1  **- ADDC sum, addend, adder**  **SUBB diff, minuend, subtrahend**  **- MOV opnd1, #*nn***  **MOV SP, #nnnn**  ; for 8-/16-bit initial-value setting  **- JUMP label**  **- JR7Z label**  **- JR7NZ label**  **- JC label**  **- JNC label**  ; unconditional/conditional code branching  **- RRC opnd1**  **- RLC opnd1**  **- CMP opnd1, opnd2** |

|  |  |
| --- | --- |
| // what is the code below  //for?  . . .  or R1, R1, #0F0H  . . .  // data manipulation at the  // level below BYTE; e.g.  // . . . clear lower-half of  // a data byte  // . . . complement every  // even-bit in a data  // byte  // . . . exchange two bits in  // a data byte  // . . . reverse bit-order in  // a data byte  - - - - - - - - - - - - - - - - - - - - - -  mov SP, 4000H  . . .  mov R0, 1000H  clr R1  push R0  and R0, R0, #1  rr R0  or R1, R0, R1 ; bit0  pop R0  push R0  and R0, R0, #2  rr R0  rr R0  rr R0  or R1, R0, R1 ; bit1  pop R0  push R0  and R0, R0, #4  rl R0 | computing capacity ( TWN#2 continued)  **- CMP opnd1, opnd2**  **CMP opnd1, #nn**  **CMP DPTRi, #nnnn**  **CMP DPTRi, m\_opnd**  - arithmetic/logic instructions involved with  Immediate-value  **- CLR C**  **CLR Z**  **CLR P**  **SET C**  **SET Z**  **SET P** |

|  |  |
| --- | --- |
| rl R0  rl R0  or R1, R0, R1 ; bit2  pop R0  push R0  and R0, R0, #8  rl R0  or R1, R0, R1 ; bit3  pop R0  push R0  and R0, R0, #10H  rr R0  or R1, R0, R1 ; bit4  pop R0  push R0  and R0, R0, #20H  rr R0  rr R0  rr R0  or R1, R0, R1 ; bit5  pop R0  push R0  and R0, R0, #40H  rl R0  rl R0  rl R0  or R1, R0, R1 ; bit6  pop R0  push R0  and R0, R0, #80H  rl R0  or R1, R0, R1 ; bit7  pop R0  mov R1m 1001H  . . .  // what is this UGLY code for? |  |