5. & 7. ARM: Architecture (3)(4)

18-349: Embedded Real-Time Systems

Priya Narasimhan
Electrical & Computer Engineering
Carnegie Mellon University

http://www.ece.cmu.edu/~ee349

Carnegie Mellon

Electrical & Computer ENGINEERING

Overview of Today's Lecture

- More ARM instructions
  - Loading and storing single registers from/to memory
  - Loading and storing multiple registers from/to memory
- Using load/store multiple instructions for stack operations

Single Register Data Transfer

- ARM is based on a "load/store" architecture
  - All operands should be in registers
  - Load instructions are used to move data from memory into registers
  - Store instructions are used to move data from registers to memory
  - Flexible – allow transfer of a word or a half-word or a byte to and from memory

```
LDR/STR        Word
LDRB/STRB      Byte
LDRH/STRH      Halfword
LDRSB          Signed byte load
LDRSH          Signed halfword load
```

- Syntax:
  - `LDR{<cond>}{<size>} Rd, <address>`
  - `STR{<cond>}{<size>} Rd, <address>`

3

LDR and STR

- `LDR` and STR instructions can load and store data on a boundary alignment that is the same as the datatype size being loaded or stored
- `LDR` can only load 32-bit words on a memory address that is a multiple of 4 bytes – 0, 4, 8, and so on

- `LDR r0, [r1]`
  - Loads register r0 with the contents of the memory address pointed to by register r1

- `STR r0, [r1]`
  - Stores the contents of register r0 to the memory address pointed to by register r1

- Register r1 is called the base address register

4

Addressing Modes

- ARM provides three addressing modes
  - Preindex with writeback
  - Preindex
  - Postindex
- Preindex mode useful for accessing a single element in a data structure
- Postindex and preindex with writeback useful for traversing an array

5

Addressing Modes

- Preindex
  - Example: `LDR r0, [r1, #4]`
- Preindex with writeback
  - Calculates address from a base register *plus* address offset
  - Updates the address in the base register with the new address
  - The *updated base register value* is the address used to access memory
  - Example: `LDR r0, [r1, #4]!`
- Postindex
  - Only updates the base register *after* the address is used
  - Example: `LDR r0, [r1], #4`

6

More on Addressing Modes

- Address `<address>` accessed by LDR/STR is specified by
  - A base register plus an offset

- Offset takes one of the three formats
  1. Immediate: offset is a number that can be added to or subtracted from the base register

     Example:
     ```
     LDR r0,[r1,  #8];        r0 ⇦ mem[r1+8]
     LDR r0,[r1, #-8];        r0 ⇦ mem[r1-8]
     ```

  2. Register: offset is a general-purpose register that can be added to or subtracted from the base register

     Example:
     ```
     LDR r0,[r1,  r2];        r0 ⇦ mem[r1+r2]
     LDR r0,[r1, -r2];        r0 ⇦ mem[r1-r2]
     ```

  3. Scaled Register: offset is a general-purpose register shifted by an immediate value and then added to or subtracted from the base register

     Example:
     ```
     LDR r0,[r1,r2, LSL #2];        r0 ⇦ mem[r1+4*r2]
     ```

7

Multiple-Register Transfer

- Load-store-multiple instructions can transfer multiple registers between memory and the processor in a single instruction
- Advantages
  - More efficient than single-register transfers for moving blocks of data around memory
  - More efficient for saving and restoring **context** and **stacks**
- Disadvantages
  - ARM does not interrupt instructions when executing ⇨ load-store multiple instructions can increase interrupt latency
- Compilers can limit interrupt latency by providing a switch to control the max number of registers that can be transferred on a load-store-multiple

```
LDM<cond><addrMode> Rn{!}, <registerList>{^}
STM<cond><addrMode> Rn{!}, <registerList>{^}
```

8

More on Load-Store-Multiple

- Transfer occurs from a base-address register `Rn` pointing into memory
- Transferred registers can be either
  - Any subset of the current bank of registers (default)
  - Any subset of the user mode bank of registers when in a privileged mode (postfix instruction with a '^')
    - Processor not in user mode or system mode
    - Writeback is not possible, i.e., `!` cannot be supported at the same time
    - If `pc` is in the list of registers, additionally copy `spsr` to `cpsr`
- Register `Rn` can be optionally updated following the transfer
  - If register `Rn` is followed by the `!` character
- Registers can be individually listed or lumped together as a range
  - Use a comma with "{" and "}" parentheses to list individual registers
  - Use a "-" to indicate a range of registers
  - Good practice to list the registers in the order of increasing register number (since this is the usual order of memory transfer)

9

Addressing Modes for Load-Store-Multiple

- Suppose that `N` is the number of registers in the list of registers
- `IA` (increment after)
  - Start reading at address `Rn`; ending address is `Rn + 4N - 4`
  - `Rn!` equals `Rn + 4N`
- `IB` (increment before)
  - Start reading at address `Rn+4`; ending address is `Rn + 4N`
  - `Rn!` equals `Rn + 4N`
- `DA` (decrement after)
  - Start reading at address `Rn - 4N + 4`; ending address is `Rn`
  - `Rn!` equals `Rn - 4N`
- `DB` (decrement before)
  - Start reading at address `Rn - 4N`; ending address is `Rn - 4`
  - `Rn!` equals `Rn - 4N`
- ARM convention: DB and DA are like loading the register list backwards from sequentially descending memory addresses

10

Things to Remember

- Any register can be used as the base register
- Any register can be in the register list
- Order of registers in the list does not matter
- The lowest register always uses the lowest memory address *regardless of the order in which registers are listed in the instruction*
- LDM and STM instructions only transfer words
  - Unlike LDR/STR instructions, they don't transfer bytes or half-words
- Can specify range instead of individual registers
  - Example: `LDMIA r10!, {r12, r2-r7}`
- If the base register is updated (using `!`) in the instruction, then it cannot be a part of the register set
  - Example: `LDMIA r10!, {r0, r1, r4, r10}` is not allowed

11

Examples

PRE
```
r0 = 0x00080010
r1 = 0x00000000
r2 = 0x00000000
r3 = 0x00000000
mem32[0x8001c] = 0x04
mem32[0x80018] = 0x03
mem32[0x80014] = 0x02
mem32[0x80010] = 0x01
```

	0x00080020	0x05
	0x0008001c	0x04
	0x00080018	0x03
	0x00080014	0x02
r0 (original) →	0x00080010	0x01
	0x0008000c	0x00

```
LDMIA r0!, {r1-r3}

POST  r0 =
      r1 =
      r2 =
      r3 =
```

```
LDMIB r0!, {r1-r3}

POST  r0 =
      r1 =
      r2 =
      r3 =
```

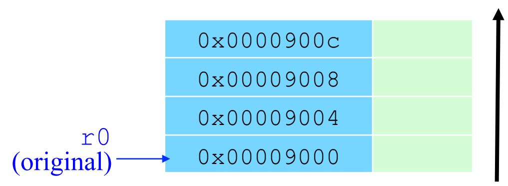
12

Example 1: Saving & Restoring Registers

- Here's what we want to accomplish
  - Save the contents of registers `r1`, `r2` and `r3` to memory
  - Mess with the contents of registers `r1`, `r2` and `r3`
  - Restore the original contents of `r1`, `r2` and `r3` from memory & restore `r0`

```
PRE     r0 = 0x00009000
        r1 = 0x09
        r2 = 0x08
        r3 = 0x07

; store contents to memory
STMIB r0!, {r1-r3}
; mess with registers r1, r2, r3
MOV r1, #1
MOV r2, #2
MOV r3, #3
; restore original r1, r2, r3
LDMDA r0!, {r1-r3}
```



ARM convention: Highest memory location maps to highest numbered register

Example 1: Block Copying

- Here's what we want to accomplish
  - Copy blocks of 32 bytes from a source address to a destination address
  - `r9` points to the start of the source data
  - `r10` points to the start of the destination data
  - `r11` points to the end of the source data

```
loop
    ; load 32 bytes from source address and update r9 pointer
    LDMIA r9!, {r0-r7}
    ; store 32 bytes to destination address and update r10 pointer
    STMIA r10!, {r0-r7}
    ; check if we are done with the entire block copy
    CMP r9, r11
    ; continue until done
    BNE loop
```

Stack Operations

- ARM uses load-store-multiple instructions to accomplish stack operations
- Pop (removing data from a stack) uses load-multiple
- Push (placing data on a stack) uses store-multiple
- Stacks are ascending or descending
  - Ascending (A): Grow towards higher memory addresses
  - Descending (D): Grow towards lower memory addresses
- Stacks can be full or empty
  - Full (F): Stack pointer `sp` points to the last used or full location
  - Empty (E): Stack pointer `sp` points to the first unused or empty location
- Four possible variants
  - Full ascending (FA) – `LDMFA` & `STMFA`
  - Full descending (FD) – `LDMFD` & `STMFD`
  - Empty ascending (EA) – `LDMEA` & `STMEA`
  - Empty descending (ED) – `LDMED` & `STMED`

15

Stacks on the ARM

- ARM has an ARM-Thumb Procedure Call Standard (ATPCS)
  - Specifies how routines are called and how registers are allocated
- Stacks according to ATPCS
  - Full descending
- What does this mean for you?
  - Use `STMFD` to store registers on stack at procedure entry
  - Use `LDMFD` to restore registers from stack at procedure exit
- What do these handy aliases actually represent?
  - `STMFD` = `STMDB` (store-multiple-decrement-before)
  - `LDMFD` = `LDMIA` (load-multiple-increment-after)

16

Example

```
PRE     r1 = 0x00000002
        r4 = 0x00000003
        sp = 0x00080014


STMFD   sp!, {r1, r4}
```

sp (original) → 0x00080014

0x00080018	0x05
0x00080014	0x04
0x00080010	*Empty*
0x0008000c	*Empty*

0x00080018	0x05
0x00080014	0x04
0x00080010	***0x03***
0x0008000c	***0x02***

sp (final) → 0x0008000c

17

Stack Checking

- Three stack attributes to be preserved
- Stack base
  - Starting address of the stack in memory
  - If `sp` goes past the stack base, stack underflow error occurs
- Stack pointer (`sp`)
  - Initially points to the stack base
  - As data is inserted when a program executes, `sp` descends memory and points to top of the stack
- Stack limit (`sl`)
  - If `sp` passes the stack limit, a stack overflow error occurs
  - ATPCS: `r10` is defined as `sl`
    - If `sp` is less than `r10` after items are pushed on the stack, stack overflow occurs

18

Overview of Next Part of Lecture

- Loading arbitrary 32-bit constants in registers
  - Pseudo instructions
  - Literal pools
- Instruction encoding
- Limitations of B/BL instructions
- SWI instructions
- Program status register instructions
- ATPCS conventions
- GNU Assembler

19

ARM Instruction Set Encoding

- Remember we said that all ARM instructions are 32 bits long?
- All ARM instructions encoded
  - Contains condition code
  - Information about whether the processor changes mode
  - Register list, a bit field where a bit is set if the corresponding register is used
  - Writeback addressing mode employed
  - And much more

- What constraints does this impose on operand lengths and on what is possible in each operation?

20

Digging Deeper: Loading Constants

- There is no single instruction which will load a 32 bit immediate constant into a register without performing a data load from memory.
  - All ARM instructions are 32 bits long
  - ARM instructions do not use the instruction stream as data
- The data processing instruction format has 12 bits available for operand2
  - If used directly, this would only give a range of 4096 bytes
- Instead it is used to store 8-bit constants, giving a range of 0 - 255
- These 8 bits can then be rotated right through an even number of positions
  - This gives a much larger range of constants that can be directly loaded, though some constants will still need to be loaded from memory

Specific Example: `MOV`

- `MOV` or `MVN` data processing instruction can be used to load 8-bit numbers into registers
  - `MOV r0, #0x07        ;r0=0x00000007`
- Use `MOV` with the barrel shifter to load more than 8-bit numbers into registers
  - `MOV r0, #0x0F, #2   ; r0=0xC0000003`
- Remember operand2 in `MOV` instruction takes 12 bits
  - 8 bits are used for the immediate value
  - 4 bit rotate value (which should be an even number between 0 and 30)
- Rule to remember is “8-bits rotated right by an even number of bit positions”.

Loading Constants: Pseudo-Instruction `LDR`

- To allow larger constants to be loaded, the assembler offers a pseudo-instruction:
  - `LDR Rd, =const`
- This will either:
  - Produce a `MOV` or `MVN` instruction to generate the value (if possible)

  or
  - Generate a `LDR` instruction with a `pc`-relative address to read the constant from a *literal pool* (constant data area embedded in the code)
  - What is a literal pool?
    - Portion of memory used by the assembly code to store constants
- This is the recommended way of loading constants into a register

23

Loading Constants – Example

Original assembly code

```
LDR r0, =0x55555555
LDR r1, =0x4867
ADD r2, r1, r0
SWI 0x123456
```

Machine code generated by assembler

```
0x00000000:    e59f0008
0x00000004:    e59f1008
0x00000008:    e0812000
0x0000000c:    ef123456
0x00000010:    55555555
0x00000014:    00004867
```

- Assembler places the constant `0x55555555` at location `0x00000010`
- and converts the pseudo-instruction to `LDR r0, [pc, #8]`
- Assembler places the constant `0x4867` at location `0x00000014`
- and converts the pseudo-instruction to `LDR r0, [pc, #8]`

24

ARM Instruction Set Encoding

Bits 31 … 0

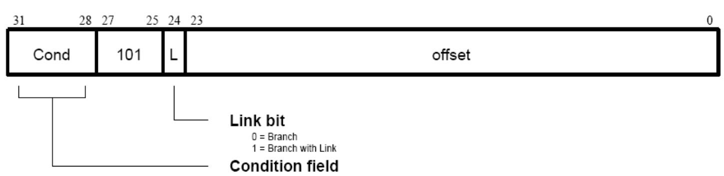
Cond | 0 | 0 | I | Opcode | S | Rn | Rd | Operand 2	*Data Processing / PSR Transfer*
Cond | 0 | 0 | 0 | 0 | 0 | 0 | A | S | Rd | Rn | Rs | 1 | 0 | 0 | 1 | Rm	*Multiply*
Cond | 0 | 0 | 0 | 0 | 1 | U | A | S | RdHi | RdLo | Rn | 1 | 0 | 0 | 1 | Rm	*Multiply Long*
Cond | 0 | 0 | 0 | 1 | 0 | B | 0 | 0 | Rn | Rd | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | Rm	*Single Data Swap*
Cond | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | Rn	*Branch and Exchange*
Cond | 0 | 0 | 0 | P | U | 0 | W | L | Rn | Rd | 0 | 0 | 0 | 0 | 1 | S | H | 1 | Rm	*Halfword Data Transfer: register offset*
Cond | 0 | 0 | 0 | P | U | 1 | W | L | Rn | Rd | Offset | 1 | S | H | 1 | Offset	*Halfword Data Transfer: immediate offset*
Cond | 0 | 1 | I | P | U | B | W | L | Rn | Rd | Offset	*Single Data Transfer*
Cond | 0 | 1 | 1 | | 1 |	*Undefined*
Cond | 1 | 0 | 0 | P | U | S | W | L | Rn | Register List	*Block Data Transfer*
Cond | 1 | 0 | 1 | L | Offset	*Branch*
Cond | 1 | 1 | 0 | P | U | N | W | L | Rn | CRd | CP# | Offset	*Coprocessor Data Transfer*
Cond | 1 | 1 | 1 | 0 | CP Opc | CRn | CRd | CP# | CP | 0 | CRm	*Coprocessor Data Operation*
Cond | 1 | 1 | 1 | 0 | CP Opc | L | CRn | Rd | CP# | CP | 1 | CRm	*Coprocessor Register Transfer*
Cond | 1 | 1 | 1 | 1 | Ignored by processor	*Software Interrupt*

Bits 31 … 0

25

Digging Deeper

- Branch instructions



  - When executing the instruction, the processor
    - Shifts the offset left two bits, sign extends it to 32 bits, and adds it to the `pc`
  - This gives a 26-bit offset from the current instruction
  - What is the maximum distance (in bytes) that we can jump to, in a branch?

26

LDR Instruction

- `LDR rd, [rn, #offset]` `rd` ← `mem[rn+offset]`

Decimal numbers prefixed by #

- `rd, rn` can be any register (`r0 – r15`)
- Binary encoding of `LDR/STR` instruction (with immediate offset addressing mode)

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
cond	0	1	0	P	U	B	W	L	Rn	Rd	offset_12

distinguish between load (1) and store (0)

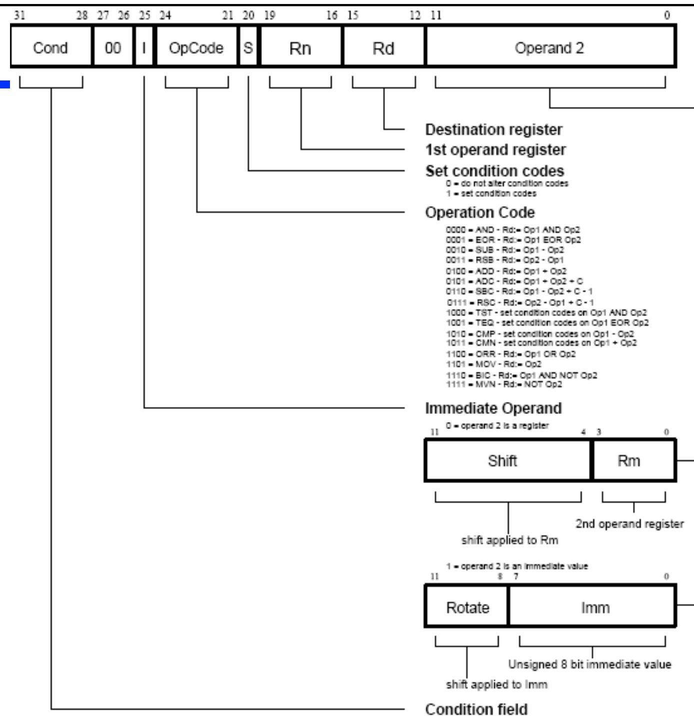
add or subtract offset

LDR (contd.)

- `LDR` can be used for branches beyond the range of `BL` instruction
  - Example: `LDR pc, [pc, #offset]` `pc` ← `mem[pc+offset]`
  - The address of the branch should be stored in memory location `curraddr +offset+8`
- Example

```
    0x10000000     add r0, r1, r2
    0x10000004     ldr pc, [pc, #4];
    0x10000008     sub r1, r2, r3
    0x1000000c     cmp r0, r1
    0x10000010     0x20000000
    …
    …
Branch_target
    0x20000000     str r5, [r13, -#4]!
    …
    …
```

Data Processing Instructions



29

`SWI` Instruction

- **S**oft**W**are **I**nterrupt (SWI) instruction causes an exception
  - Executes in the privileged mode
  - Provides a way for applications to call operating system routines
- Similar to a sub-routine call because
  - Parameters and return values passed through registers
- Differs from a sub-routine call because the ARM processor
  - Stashes away user `cpsr`
  - Switches to supervisor mode
  - Starts executing from a specific location (typically 0x08)

30

Software Interrupt (SWI)

- In effect, a SWI is a user-defined instruction
- It causes an exception trap to the SWI hardware vector
  - Causing a change to supervisor mode, plus the associated state saving
  - Causing the SWI exception handler to be called
- The handler can then examine the comment field of the instruction to decide what operation has been requested
- By making use of the SWI mechanism, an operating system can implement a set of privileged operations which applications running in user mode can request
  - This is how system calls in the OS are implemented

31

Program Status Register Instructions

- Two instructions to directly control a Program Status Register
- MRS
  - Transfers the contents of either the `cpsr` or the `spsr` into a register
- MSR
  - Transfers the contents of a register into the `cpsr` or the `spsr`

```
PRE    cpsr = nzcvqIFt_SVC

   MRS r1, cpsr
   BIC  r1, r1, #0x080
   MSR cpsr, r1

POST   cpsr = nzcvqiFt_SVC
```

32

ARM is a RISC Architecture

- ARM conforms to the Reduced Instruction Set Computer (RISC) architecture
- Typical of RISC systems
  - A large set of general-purpose registers that can hold either data or an address
    - Registers act as the fast local memory for all data processing operations
  - Fixed-length instructions that enable pipelining
  - Load/Store model for data processing
    - Operations on registers and not directly on memory
    - All data must be loaded into registers before they can be operated on
    - What's the advantage?
  - Small number of addressing modes
    - All load/store addresses determined from registers or instruction fields
- ARM is not a pure RISC architecture
  - In one way, this is its strength – it does not take the RISC concept too far
  - Why diverge from RISC? ARM was born to support embedded systems

33

Non-RISC Aspects

- Variable cycle execution for certain instructions
  - Some instructions (load-store-multiple) vary in the number of cycles depending on the number of registers involved
  - Performance features: Load-store-multiple can occur on sequential addresses (faster than random accesses)
  - Code density also improved since multiple register transfers are often at the start and the end of functions
- Inline barrel shifter, supporting more complex operations
  - Barrel shifter is a hardware component that preprocesses one of the input registers before it is used by an instruction
  - Again, improves performance and density
- 16-bit Thumb instruction set
  - Permits the ARM processor to execute either 16- or 32-bit instructions
  - Can improve code density significantly over the 32-bit ARM instructions

34

Non-RISC Aspects

- Conditional execution
  - Instruction executed only when a specific condition has been satisfied
  - Improves performance and code density by reducing the number of branch instructions
  - Improves the flushing of pipelines
- Enhanced instructions
  - DSP (digital signal processing) instructions
  - Support fast multiplier operations
  - In some cases, an ARM processor can replace a traditional processor + DSP coprocessor and do things faster
- Auto-increment and auto-decrement addressing modes
  - Improves execution of program loops

35

ATPCS

- ARM-Thumb Procedure Call Standard (ATPCS)
  - Developed by ARM
  - Defines constraints on the use of registers
  - Defines argument-passing and result-return conventions
- Compiler-generated code conforms to the ATPCS notation
  - Coding standard that becomes important to bear in mind when mixing C and assembly
- Who saves the registers?
  - A calling routine must preserve the contents of `r0-r3` if it needs them again
  - A called routine must preserve the contents of `r4-r11` and must restore their values before returning, if it has used them
  - A called routine does not need to restore `r12` before returning
  - The value held in `r13` (`sp`) on exit from a function should be the same as it was on entry
    - `r13` should not be used for any other purpose
  - Register `r14` is the link register that contains the return address back from the function

36

Registers in ATPCS

Register	Synonym	Special	Role in the procedure call standard
r15	-	pc	Program counter.
r14	-	lr	Link register.
r13	-	sp	Stack pointer.
r12	-	ip	Intra-procedure-call scratch register.
r11	v8	-	ARM-state variable register 8.
r10	v7	sl	ARM-state variable register 7. Stack limit pointer in stack-checked variants.
r9	v6	sb	ARM-state variable register 6. Static base in RWPI variants.
r8	v5	-	ARM-state variable register 5.
r7	v4	-	Variable register 4.
r6	v3	-	Variable register 3.
r5	v2	-	Variable register 2.
r4	v1	-	Variable register 1.
r3	a4	-	Argument/result/scratch register 4.
r2	a3	-	Argument/result/scratch register 3.
r1	a2	-	Argument/result/scratch register 2.
r0	a1	-	Argument/result/scratch register 1.

37

GNU Assembler (*gas*)

- Used to produce assembler output
- Has multiple directives
  - .ascii “<string>”
    - Inserts the string as data into the assembly code
  - .align <number>
    - Aligns the address to 2^<number> bytes
  - .global <symbol>
    - Gives the symbol external linkage
  - .section <section name>
    - Starts a new code or data section: .rodata, .text, etc
  - .word <word1> ….
    - Inserts a list of 32-bit word values as data into the assembly code

38

Equivalents (C to Assembly)

```
int main (void)
{
   int a = 1024;
   int c = a + 1;
   printf (“Sum is %d\n”, c);
}
```

```
   .file “hello.c”
   .section .rodata
   .align 2

.LC0
   .ascii “Sum is %d\n”
   .text
   .align 2
   .global main

main
   MOV r3, #1024
   ADD r3, r3, #1
   MOV r1, r3
   LDR r0, .L2
   BL printf

.L2
   .word .LC0
```

39

Overview of Today’s Lecture

- Loading and storing single registers from/to memory
- Loading and storing multiple registers from/to memory
- Using load/store multiple instructions for stack operations

- Instruction set encoding and implications
- SWI instruction
- Program register instructions
- ATPCS conventions
- The GNU assembler

40