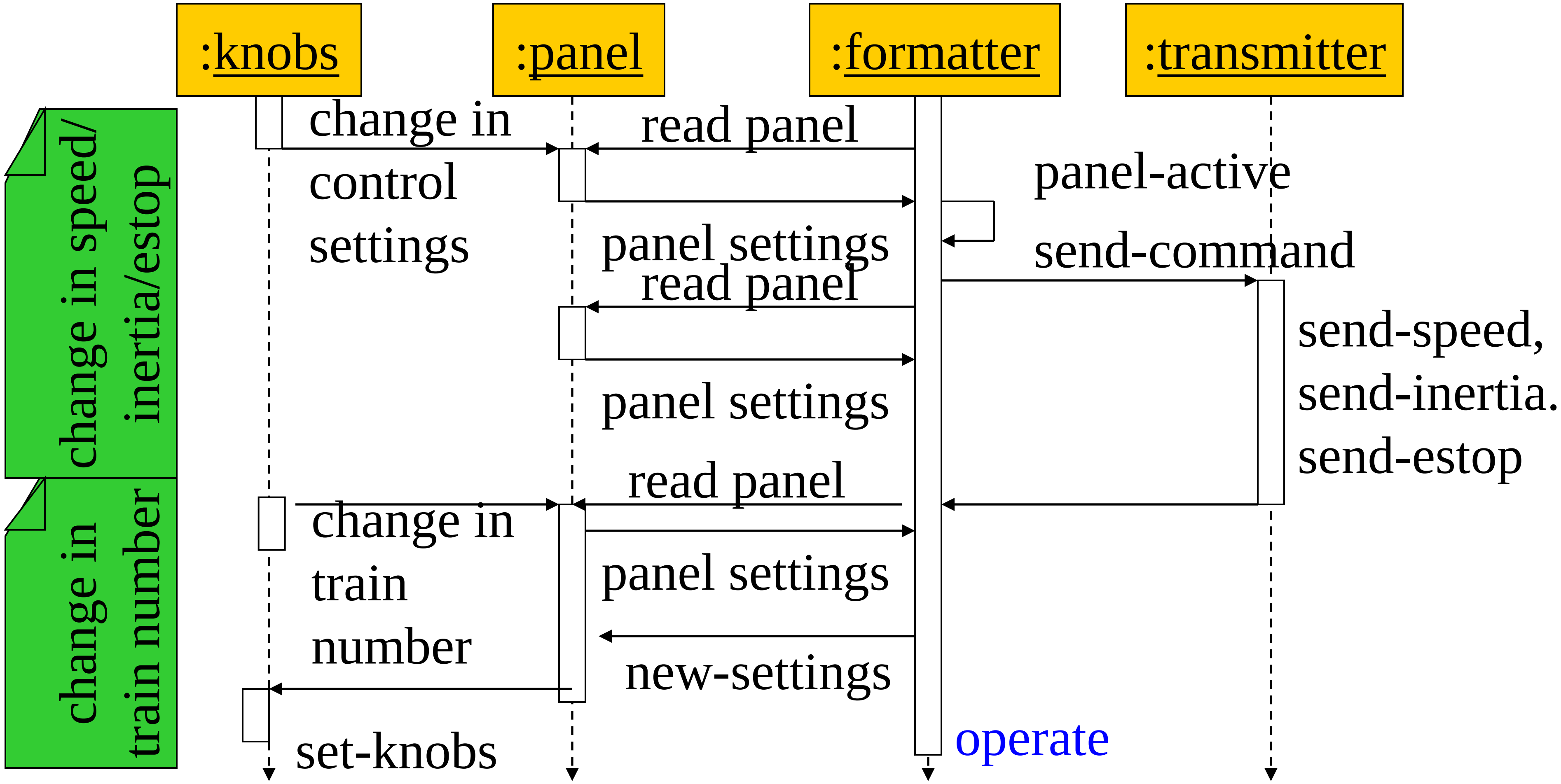
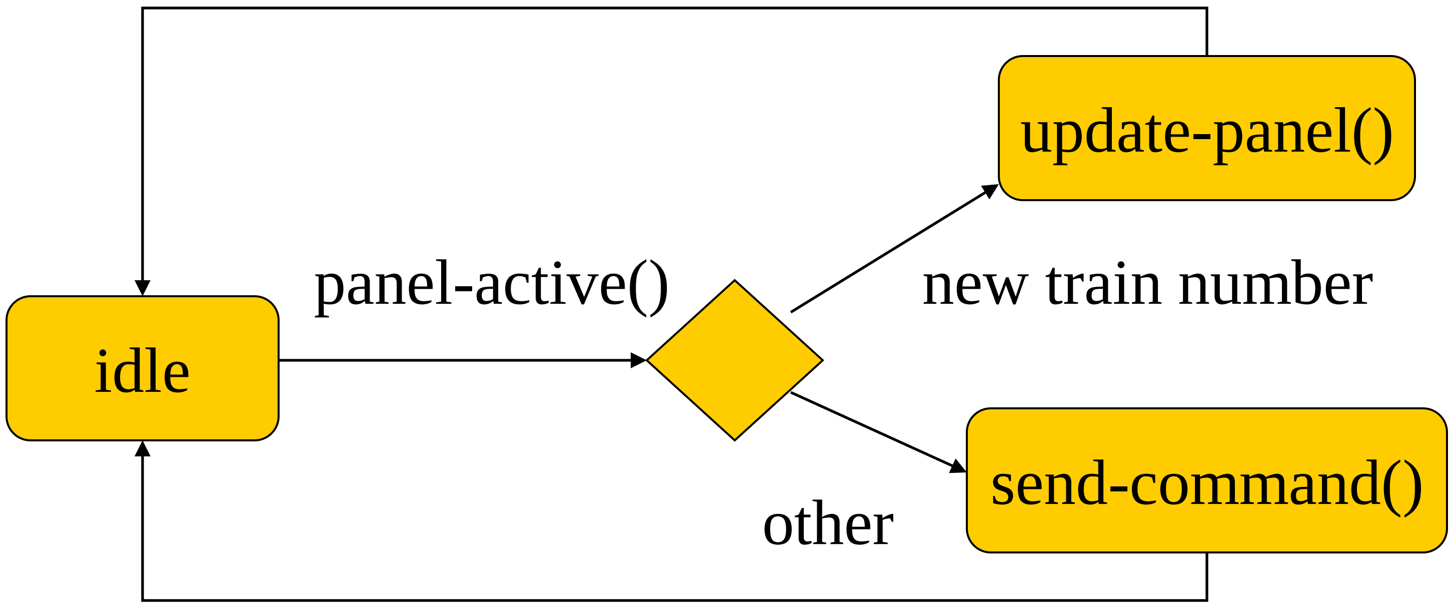
Control input cases

- Use a soft panel to show current panel settings for each train.
- Changing train number:
  - must change soft panel settings to reflect current train’s speed, etc.
- Controlling throttle/inertia/estop:
  - read panel, check for changes, perform command.

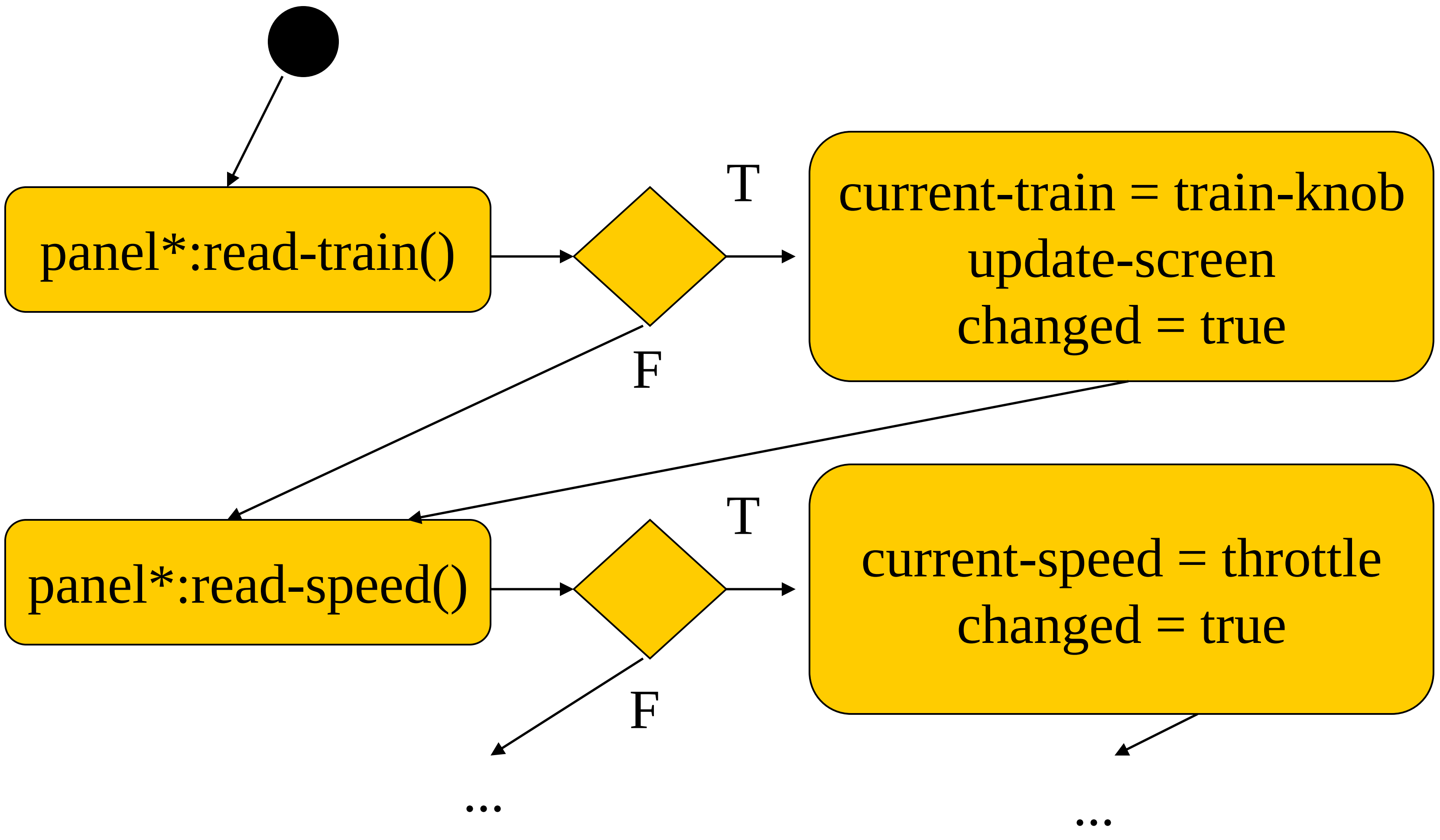
Control input sequence diagram



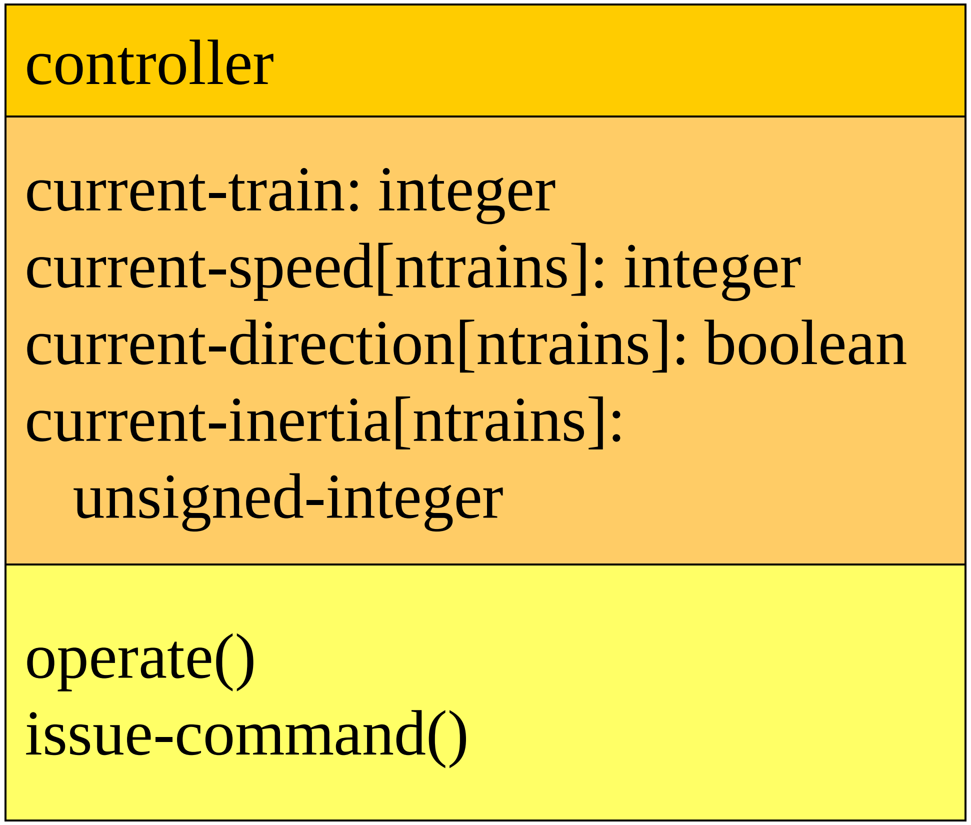
Formatter operate behavior



Panel-active behavior



Controller class

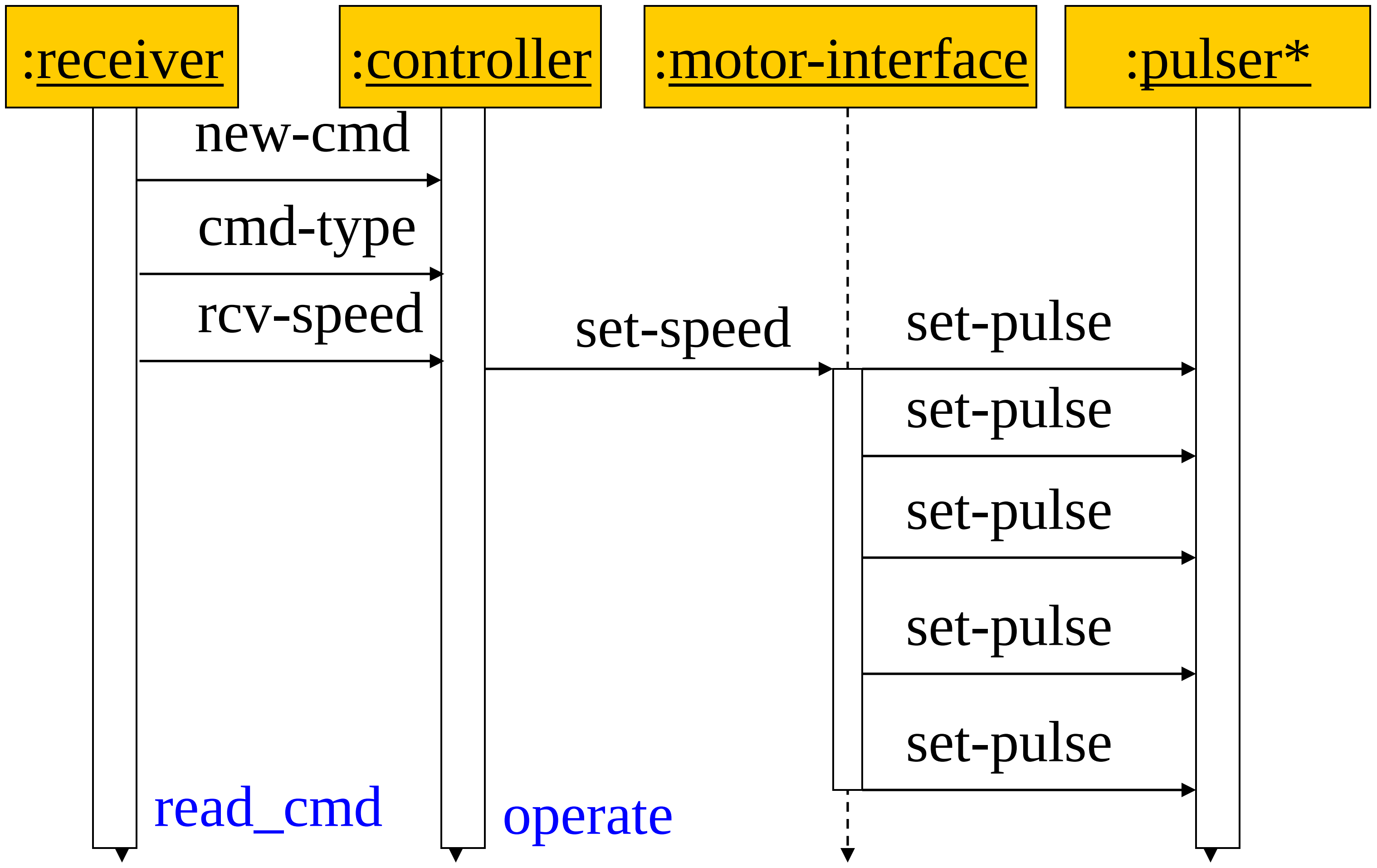


```
controller
----------------------------------------
current-train: integer
current-speed[ntrains]: integer
current-direction[ntrains]: boolean
current-inertia[ntrains]:
   unsigned-integer
----------------------------------------
operate()
issue-command()
```

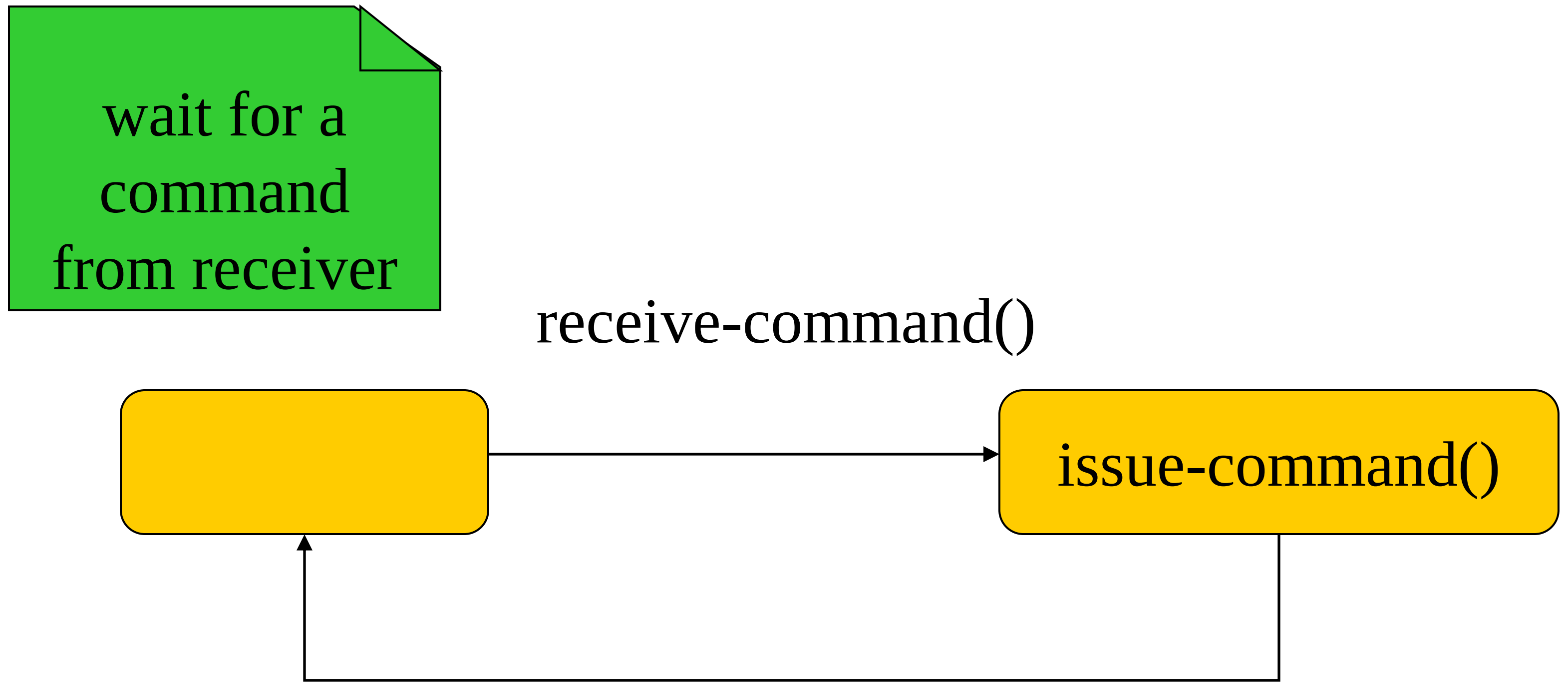
Setting the speed

- Don’t want to change speed instantaneously.
- Controller should change speed gradually by sending several commands.

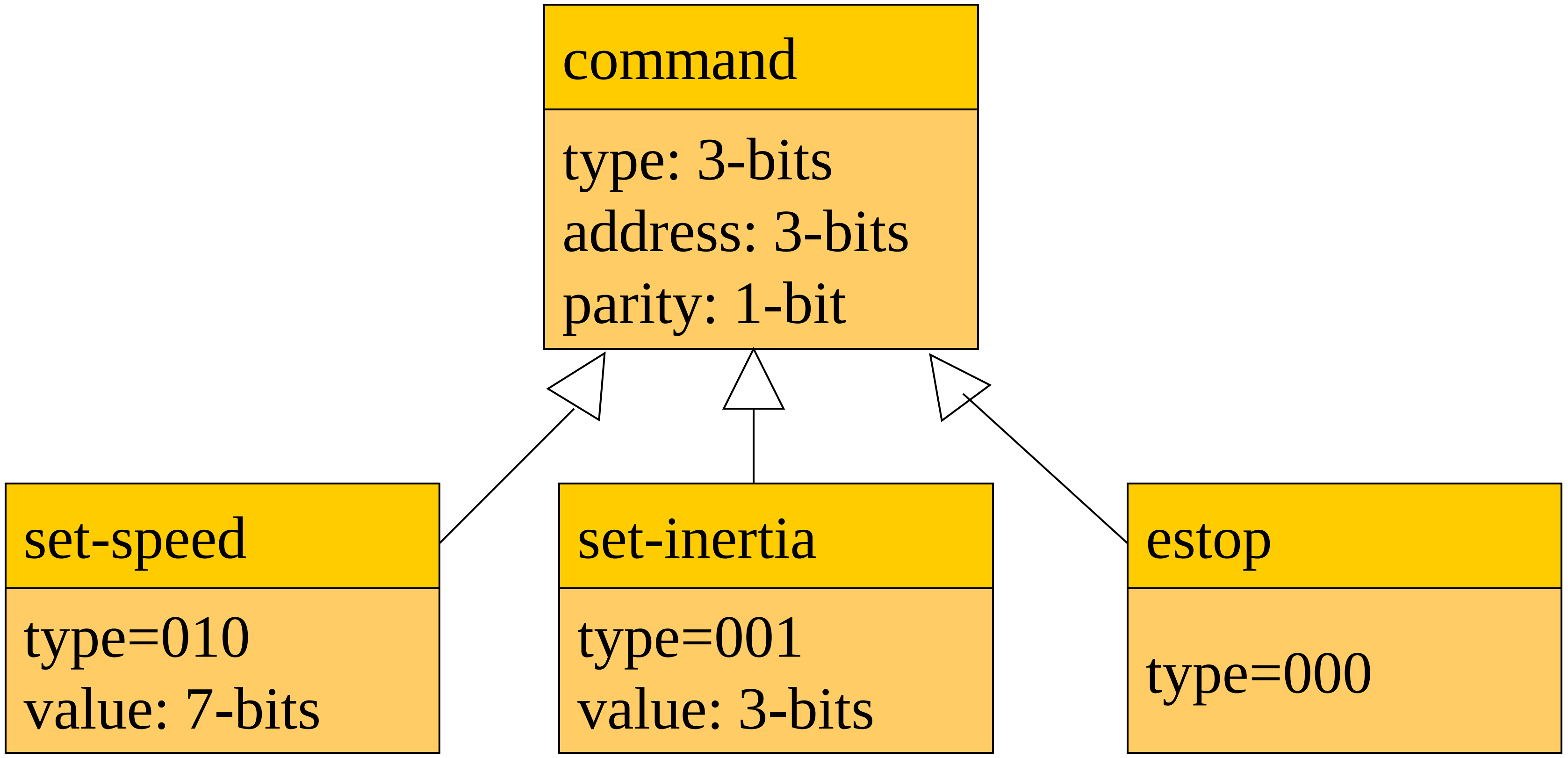
Sequence diagram for a set-speed command



Controller operate behavior



Refined command classes



Summary

- Separate specification and programming.
  - Small mistakes are easier to fix in the spec.
  - Big mistakes in programming cost a lot of time.
- You can’t completely separate specification and architecture.
  - Make a few tasteful assumptions.

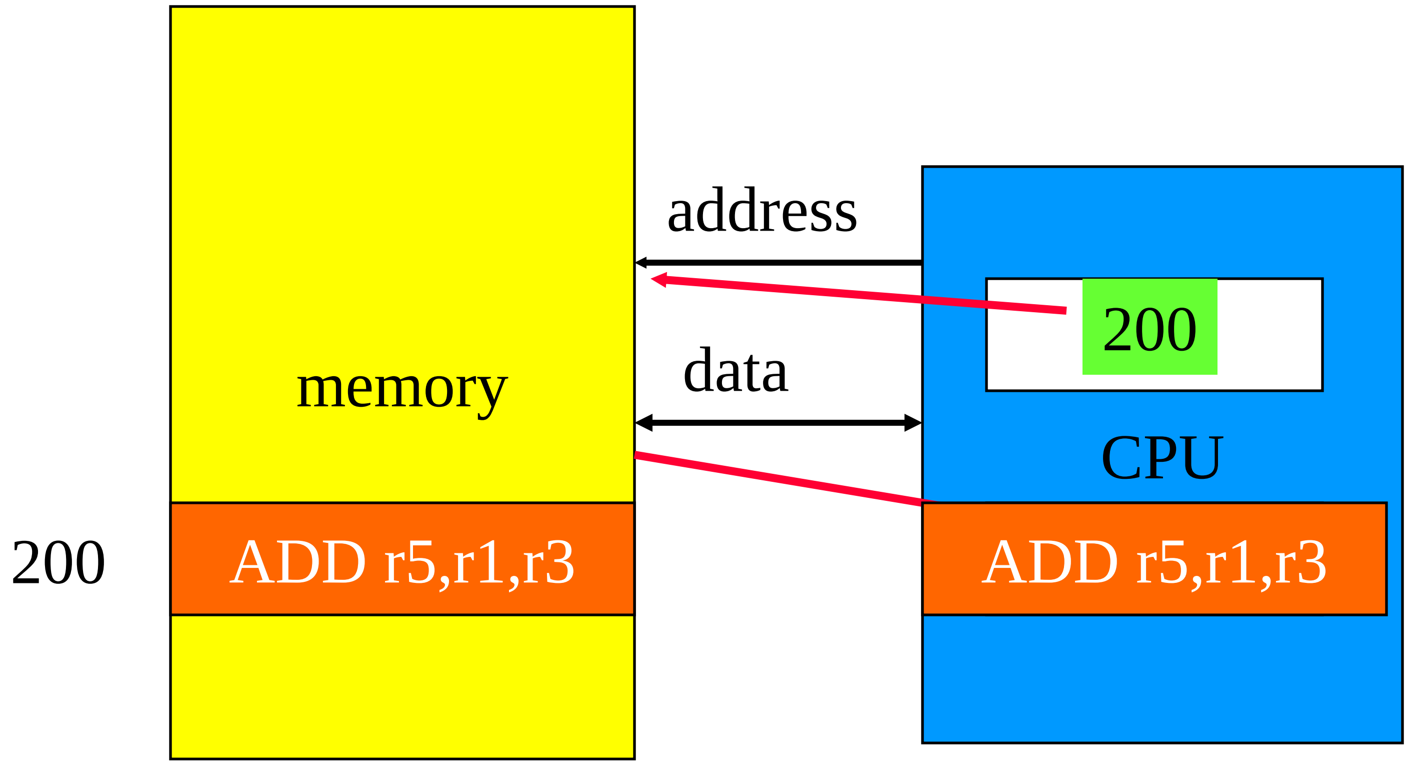
Chapter 2. Instruction sets

- Computer architecture taxonomy.
- Assembly language.

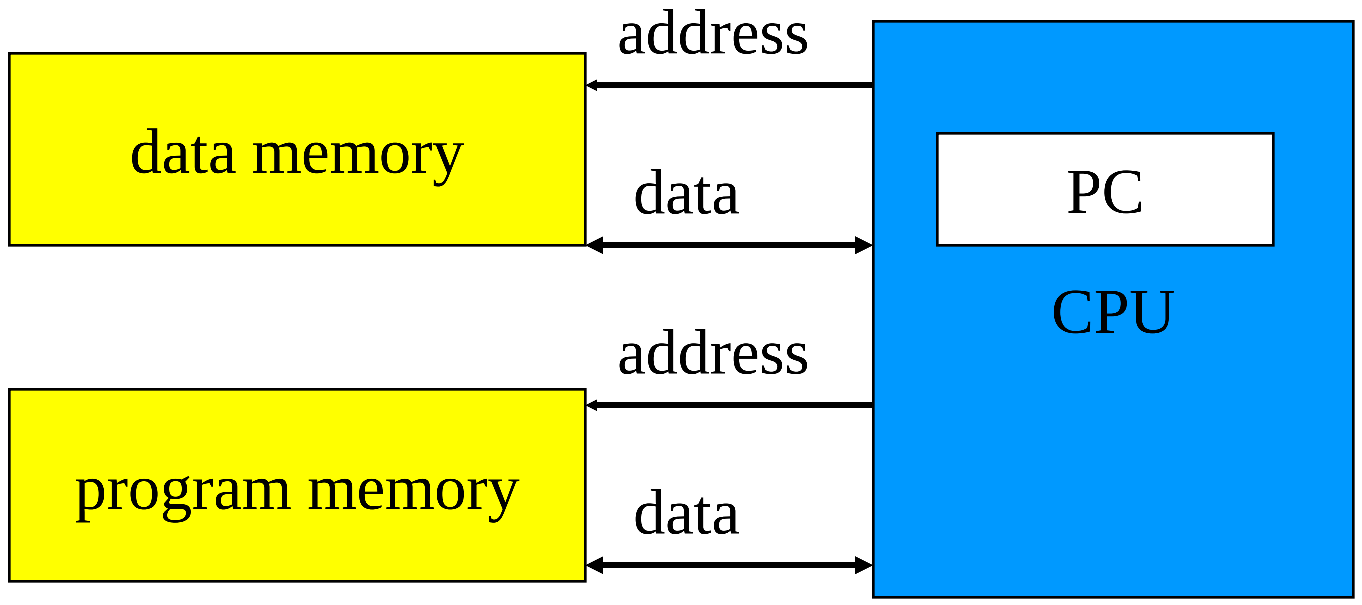
von Neumann architecture

- Memory holds data, instructions.
- Central processing unit (CPU) fetches instructions from memory.
    - Separate CPU and memory distinguishes programmable computer.
- CPU registers help out: program counter (PC), instruction register (IR), general-purpose registers, etc.

CPU + memory



Harvard architecture



von Neumann vs. Harvard

- Harvard can’t use self-modifying code.
- Harvard allows two simultaneous memory fetches.
- Most DSPs use Harvard architecture for streaming data:
  - greater memory bandwidth;
  - more predictable bandwidth.

RISC vs. CISC

- Complex instruction set computer (CISC):
  - many addressing modes;
  - many operations.
- Reduced instruction set computer (RISC):
  - load/store;
  - caches
  - pipelined instructions.

Instruction set characteristics

- Fixed vs. variable length.
- Addressing modes.
- Number of operands.
- Types of operands.

Programming model

- Or Programmer model: the set of registers visible to the programmer.
- Some registers are not visible (IR).

Multiple implementations

- Each successful architecture has several implementations:
  - varying clock speeds;
  - different bus widths;
  - different cache sizes;
  - Differenc technologies.

Assembly language

- One-to-one with instructions (more or less).
- Basic features:
  - One instruction per line.
  - Labels provide names for addresses (usually in first column).
  - Instructions often start in later columns.
  - Columns run to end of line.

Assembler

- Parse the assembly program into a machine language
- Has the structured form to make it easy to parse the program and to consider most aspects of the program line by line
- ADDGT r0, r3, #5
  - Condition, opcode, operands
  - R_destination, R_source1, R_source2

ARM assembly language example

```
label1  ADR r4,c
        LDR r0,[r4] ; a comment
        ADR r4,d
        LDR r1,[r4]
        SUB r0,r0,r1 ; comment
```

Pseudo-ops

- Some assembler directives don't correspond directly to instructions:
  - Define current address.
  - Reserve storage.
  - Constants.

ARM instruction set

- ⌘ARM versions.
- ⌘ARM assembly language.
- ⌘ARM programming model.
- ⌘ARM memory organization.
- ⌘ARM data operations.
- ⌘ARM flow of control.

ARM versions

- ARM architecture has been extended over several versions.
- We will concentrate on ARM7.

Main features of the ARM Instruction Set

- All instructions are 32 bits long.
- Most instructions execute in a single cycle.
- Every instruction can be conditionally executed.
- Instruction set extension via coprocessors

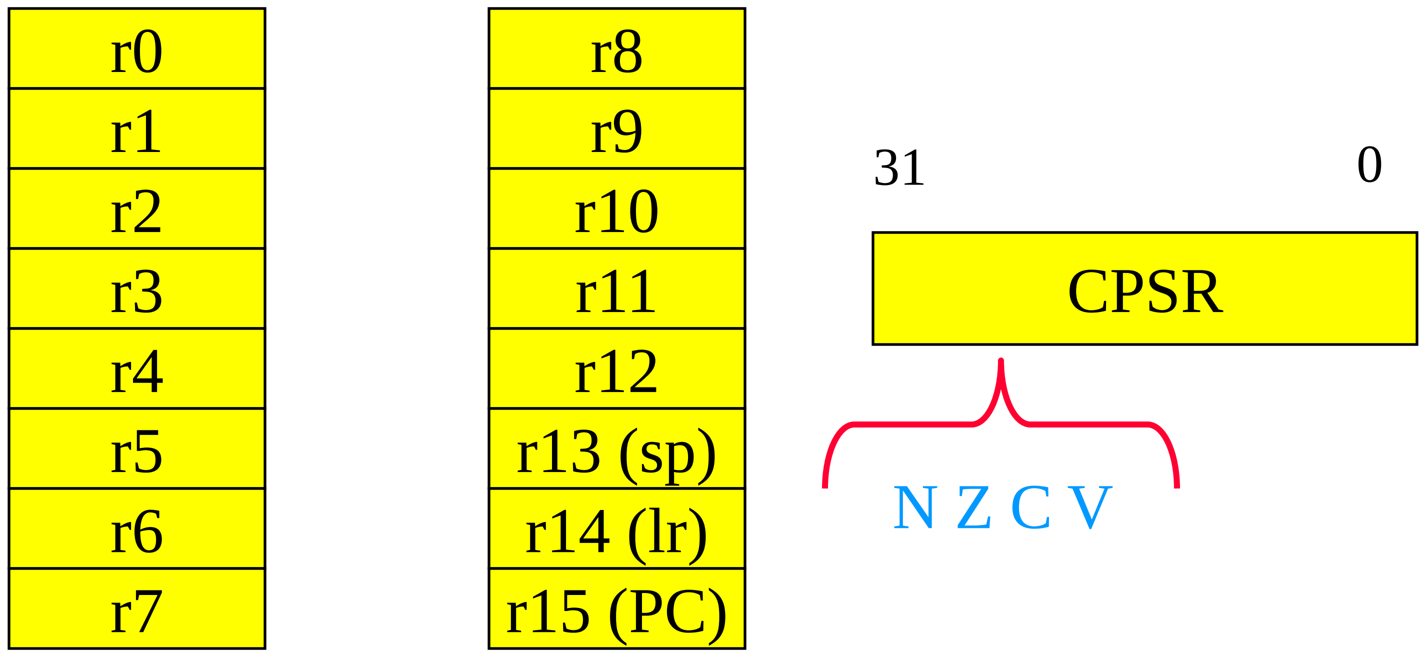
Main features of the ARM Instruction Set

- A load/store architecture
  - Data processing instructions act only on registers
    - Three operand format
    - Combined ALU and shifter for high speed bit manipulation
  - Specific memory access instructions with powerful auto-indexing addressing modes.
    - 32 bit and 8 bit data types and also 16 bit data types on ARM Architecture v4.
    - Flexible multiple register load and store instructions

Processor Modes

- The ARM has six operating modes:
  - *User* (unprivileged mode under which most tasks run)
  - *FIQ* (entered when a high priority (fast) interrupt is raised)
  - *IRQ* (entered when a low priority (normal) interrupt is raised)
  - *Supervisor* (entered on reset and when a Software Interrupt instruction is executed)
  - *Abort* (used to handle memory access violations)
  - *Undef* (used to handle undefined instructions)

ARM programming model



The 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

The Registers

- However these are arranged into several banks, with the accessible bank being governed by the processor mode. Each mode can access
  - a particular set of r0-r12 registers
  - a particular r13 (the stack pointer) and r14 (link register)
  - r15 (the program counter)
  - cpsr (the current program status register)

  and privileged modes can also access
  - a particular spsr (saved program status register)

Register Organisation

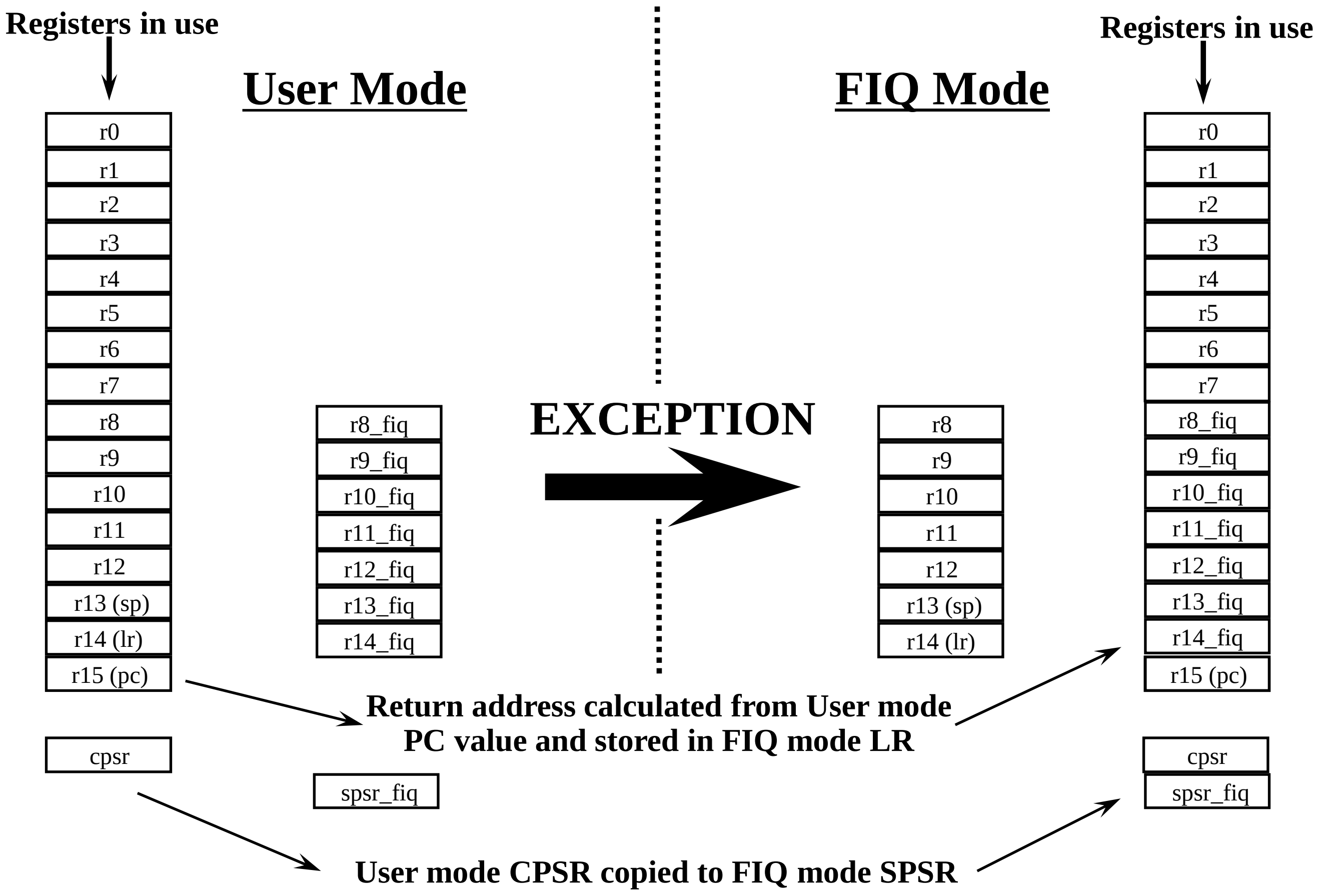
General registers and Program Counter

User32 / System	FIQ32	Supervisor32	Abort32	IRQ32	Undefined32
r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7
r8	r8_fiq	r8	r8	r8	r8
r9	r9_fiq	r9	r9	r9	r9
r10	r10_fiq	r10	r10	r10	r10
r11	r11_fiq	r11	r11	r11	r11
r12	r12_fiq	r12	r12	r12	r12
r13 (sp)	r13_fiq	r13_svc	r13_abt	r13_irq	r13_undef
r14 (lr)	r14_fiq	r14_svc	r14_abt	r14_irq	r14_undef
r15 (pc)	r15 (pc)	r15 (pc)	r15 (pc)	r15 (pc)	r15 (pc)

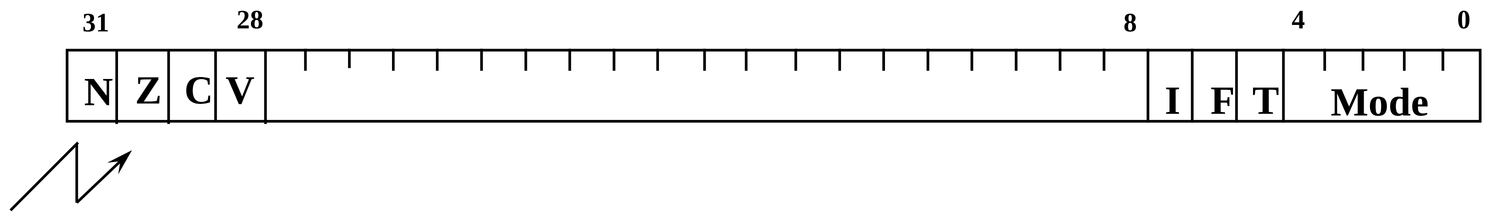
Program Status Registers

cpsr	cpsr	cpsr	cpsr	cpsr	cpsr
	spsr_fiq	spsr_svc	spsr_abt	spsr_irq	spsr_undef

Register Example: User to FIQ Mode



The Program Status Registers (CPSR and SPSRs)



Copies of the ALU status flags (latched if the instruction has the "S" bit set).

- **Condition Code Flags**

  N = **N**egative result from ALU flag.
  Z = **Z**ero result from ALU flag.
  C = ALU operation **C**arried out
  V = ALU operation o**V**erflowed

- **Mode Bits**
  M[4:0] define the processor mode.

- **Interrupt Disable bits.**
  I = 1, disables the IRQ.
  F = 1, disables the FIQ.

- **T Bit (Architecture v4T only)**
  T = 0, Processor in ARM state
  T = 1, Processor in Thumb state

Condition Flags

Flag	Logical Instruction	Arithmetic Instruction
Negative (N=‘1’)	No meaning	Bit 31 of the result has been set Indicates a negative number in signed operations
Zero (Z=‘1’)	Result is all zeroes	Result of operation was zero
Carry (C=‘1’)	After Shift operation ‘1’ was left in carry flag	Result was greater than 32 bits
oVerflow (V=‘1’)	No meaning	Result was greater than 31 bits Indicates a possible corruption of the sign bit in signed numbers

The Program Counter (R15)

- When the processor is executing in ARM state:
  - All instructions are 32 bits in length
  - All instructions must be word aligned
  - Therefore the PC value is stored in bits [31:2] with bits [1:0] equal to zero (as instruction cannot be halfword or byte aligned).

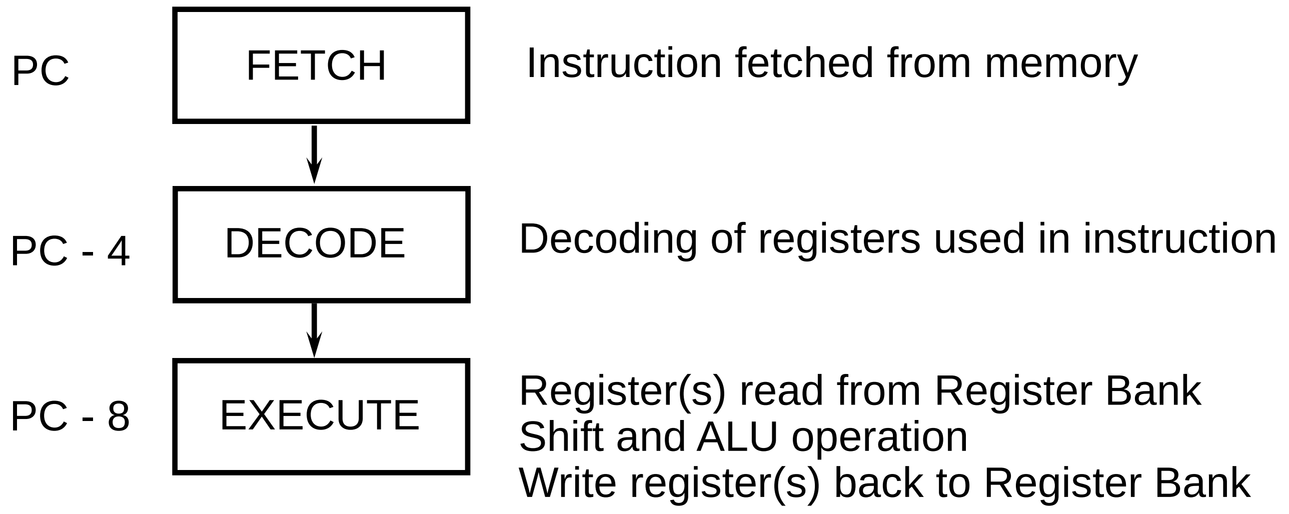
The Link Register (R14)

- R14 is used as the subroutine link register (LR) and stores the return address when Branch with Link operations are performed, calculated from the PC.
- Thus to return from a linked branch
  - `MOV r15,r14`

  or
  - `MOV pc,lr`

The Instruction Pipeline

- The ARM uses a pipeline in order to increase the speed of the flow of instructions to the processor.
  - Allows several operations to be undertaken simultaneously, rather than serially.

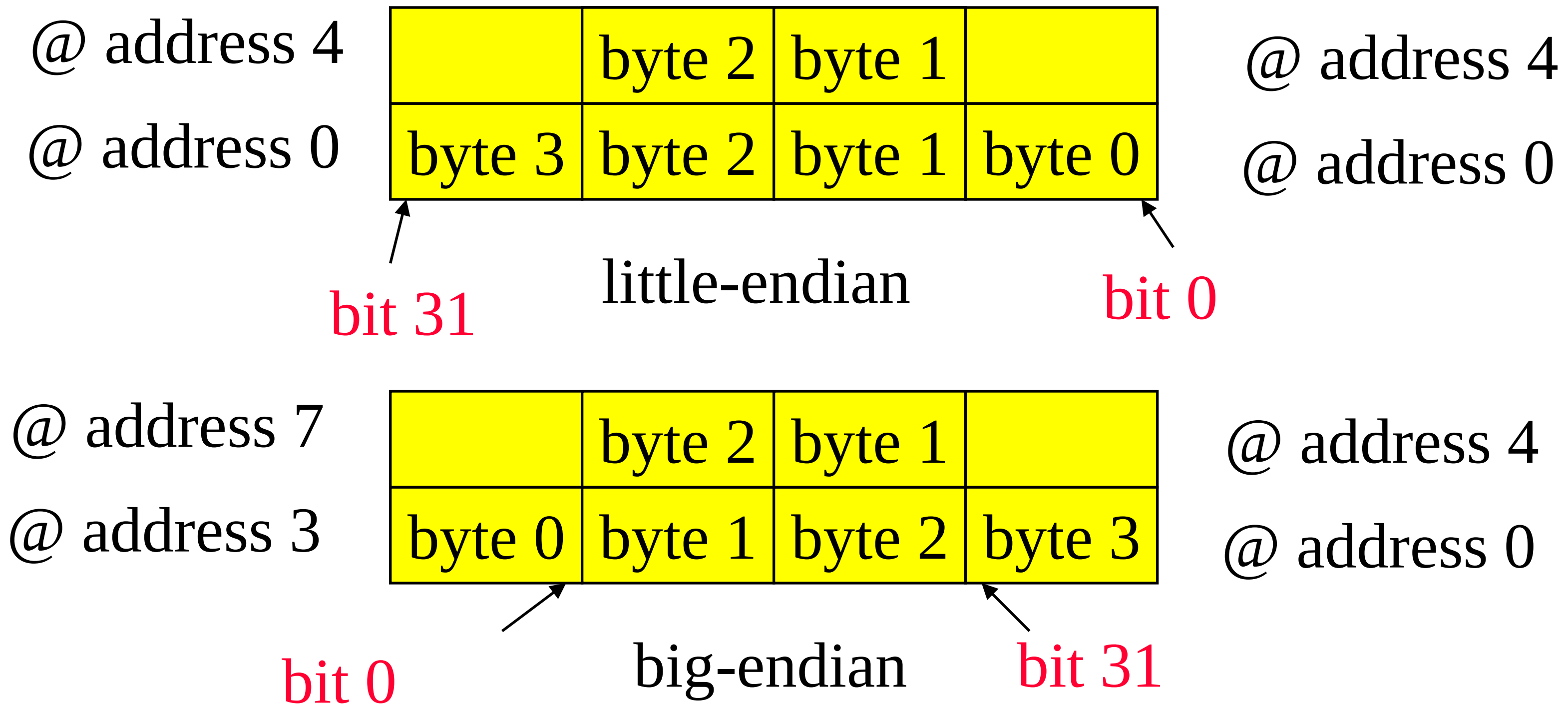

PC
FETCH
Instruction fetched from memory
PC - 4
DECODE
Decoding of registers used in instruction
PC - 8
EXECUTE
Register(s) read from Register Bank
Shift and ALU operation
Write register(s) back to Register Bank

ARM data types

- Word is 32 bits long.
- Word can be divided into four 8-bit bytes.
- ARM addresses cam be 32 bits long.
- Byte addressing: address refers to a byte.
  - A new word starts at an address of 4 multiple.
- Can be configured at power-up as either little- or bit-endian mode.

Endianness of 32-bit processors

- Relationship between bit and byte/word ordering defines endianness:



ARM status bits

- Every arithmetic, logical, or shifting operation automatically sets current program status register (CPSR) bits:
  - N (negative), Z (zero), C (carry), V (overflow).
- Examples:
  - -1 + 1 = 0: NZCV = 0110.
  - 2^{31}-1+1 = -2^{31}: NZCV = 0101.

ARM assembly language

- Fairly standard assembly language:

```
        LDR r0,[r8] ; a comment
label   ADD r4,r0,r1
```

ARM data instructions

- Basic format:

```
ADD r0,r1,r2
```

  - Computes r1+r2, stores in r0.
- Immediate operand:

```
ADD r0,r1,#2
```

  - Computes r1+2, stores in r0.

ARM data instructions

- ADD, ADC : add (w. carrry)
- SUB, SBC : subtract (w. carry)
- RSB, RSC : reverse subtract (w. carry)
- MUL, MLA : multiply (and accumulate)
- AND, ORR, EOR
- BIC : bit clear
- LSL, LSR : logical shift left/right
- ASL, ASR : arithmetic shift left/right
- ROR : rotate right
- RRX : rotate right extended with C

Subtract instructions

- SUB : subtract
  - x – y = x + ~y + 1
  - subs r0, r0 #1 ; r0=-1, setting flags

    sub r0, r1, r1, LSL #2 ; r0 = -3*r1
- SBC : subtract with carry
  - x – y - ~C = x + ~y + C
  - a 64-bit subtract: (r1,r0) – (r3,r2)

```
subs r0, r0, r2    ; subtract low word, C=NOT(borrow)
sbc  r1, r1, r3    ; subtract high words and borrow
```

Subtract instructions

- RSB : reverse subtract
  - rsb r0, r0, #0 ; r0 = - r0

    rsb r0, r1, r1, LSL#3 ; r0 = -7*r1
- RSC : reverse subtract with carry
  - negate a 64-bit integer (r1, r0)

    rsbs r0, r0, #0 ; r0 = - r0, C=NOT(borrow)

    rsc r1, r1, #0 ; r1 = - r1 - borrow

Multiply instructions

- MUL : multiply
  - mul rd, rm, rs ; rd = rm*rs
- MLA : multiply and accumulate
  - mul rd, rm, rs, rn ; rd = (rm*rs) +rn
- SMUL : signed multiply long
  - smul rdh, rdl, rm, rs ; {rdh, rdl} = rm*rs
- UMUL : signed multiply long
  - umul rdh, rdl, rm, rs ; {rdh, rdl} = rm*rs
- UMAL, SMAL
  - {rdh, rdl} = {rdh, rdl} + rm*rs

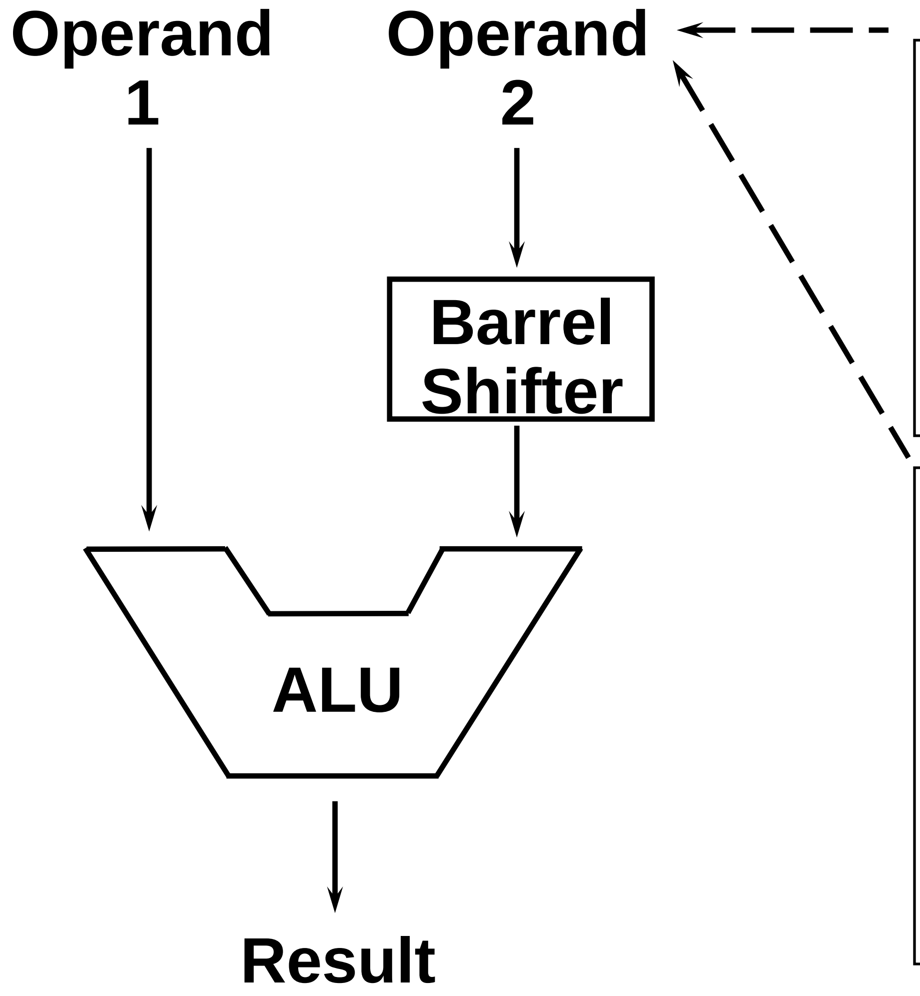
Logical instructions

- AND : logical bitwise AND of two 32-bit values
  - Rd = Rn & Rs
- ORR : logical bitwise OR of two 32-bit values
  - Rd = Rn | Rs
- EOR : logical exclusive OR of two 32-bit values
  - Rd = Rn ^ Rs
- BIC : logical bit clear (AND NOT)
  - Rd = Rn & ~Rs

Move instructions

- MOV : move a 32-bit value into a register
  - mov r7, r5 ; r7 = r5
- MVN : move the NOT of a 32-bit value into a register
  - mov r7, r5 ; r7 = ~r5

Using the Barrel Shifter



- Register, optionally with shift operation applied.
- Shift value can be either be:
  - 5 bit unsigned integer
  - Specified in bottom byte of another register.

* **Immediate value**
  - 8 bit number
  - Can be rotated right through an even number of positions.
  - Assembler will calculate rotate for you from constant.

Barrel shift operations

- LSL, LSR : logical shift left/right
    - movs r0, r1, LSL #1
- ASR : arithmetic shift right
- ROR : rotate right
- RRX : rotate right extended with C

Data operation varieties

- Logical shift:
  - fills with zeroes.
- Arithmetic shift:
  - fills with the sign bit.
- RRX performs 33-bit rotate, including C bit from CPSR above sign bit.

Comparison/test instructions

- CMP : compare
- CMN : negated compare
- TST : bit-wise test with a value
- TEQ : bit-wise test for two values
- These instructions set only the NZCV bits of CPSR.

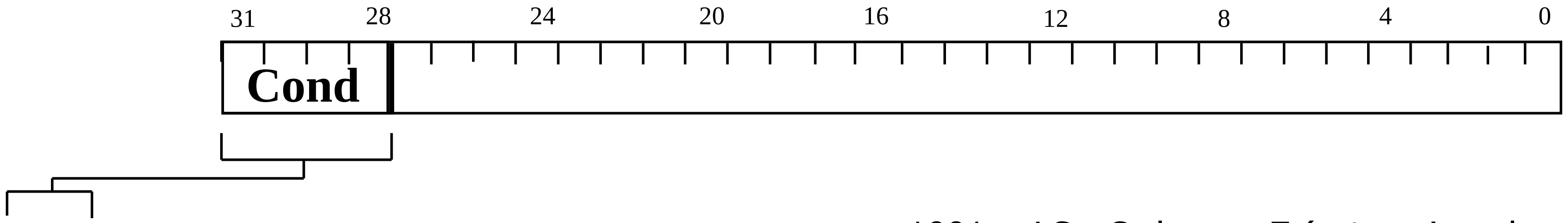
Conditional Execution

- Most instruction sets only allow branches to be executed conditionally.
- However by reusing the condition evaluation hardware, ARM effectively increases number of instructions.
  - All instructions contain a condition field which determines whether the CPU will execute them.
  - Non-executed instructions soak up 1 cycle.
    - Still have to complete cycle so as to allow fetching and decoding of following instructions.

Conditional Execution

- This removes the need for many branches, which stall the pipeline (3 cycles to refill).
  - Allows very dense in-line code, without branches.
  - The Time penalty of not executing several conditional instructions is frequently less than overhead of the branch
    or subroutine call that would otherwise be needed.

The Condition Field



0000 = EQ - Z set (equal)

0001 = NE - Z clear (not equal)

0010 = HS / CS - C set (unsigned higher or same)

0011 = LO / CC - C clear (unsigned lower)

0100 = MI -N set (negative)

0101 = PL - N clear (positive or zero)

0110 = VS - V set (overflow)

0111 = VC - V clear (no overflow)

1000 = HI - C set and Z clear (unsigned higher)

1001 = LS - C clear or Z (set unsigned lower or same)

1010 = GE - N set and V set, or N clear and V clear (>or =)

1011 = LT - N set and V clear, or N clear and V set (>)

1100 = GT - Z clear, and either N set and V set, or N clear and V set (>)

1101 = LE - Z set, or N set and V clear,or N clear and V set (<, or =)

1110 = AL - always

1111 = NV - reserved.

Comparison instructions

- CMP : compare two 32-bit integers
  - CMP r0, r1, LSR#2; compare r0 with (r1/4)
    BHS label ; if r0 >= (r1/4) goto label
- CMN : compare negative
  - CMN r0, #3; ; compare r0 with (-3)
    BLT label ; if r0 < (-3) goto label

Test instructions

- TST : bit-wise test of a 32-bit value
  - TST r0,#0xFF ; test if the LSB 8 bits are 0
- TEQ : bit-wise test for two 32-bit values
  - TEQ r0, #1 ; test to see if r0==1

Load/store instructions

- LDR, LDRH, LDRB : load (half-word, byte)
- STR, STRH, STRB : store (half-word, byte)
- Addressing modes:
  - register indirect : `LDR r0,[r1]`
  - with second register : `LDR r0,[r1,-r2]`
  - with constant : `LDR r0,[r1,#4]`

ADR pseudo-op

- Cannot refer to an address directly in an instruction.
- Generate value by performing arithmetic on PC.
- ADR pseudo-op generates instruction required to calculate address:

```
ADR r1,FOO
```

Example: C assignments

- C:

```
x = (a + b) - c;
```

- Assembler:

```
ADR r4,a          ; get address for a
LDR r0,[r4]       ; get value of a
ADR r4,b          ; get address for b, reusing r4
LDR r1,[r4]       ; get value of b
ADD r3,r0,r1      ; compute a+b
ADR r4,c          ; get address for c
LDR r2[r4]        ; get value of c
```

C assignment, cont'd.

```
SUB r3,r3,r2      ; complete computation of x
ADR r4,x          ; get address for x
STR r3[r4]        ; store value of x
```

Example: C assignment

- C:

```
y = a*(b+c);
```

- Assembler:

```
ADR r4,b ; get address for b
LDR r0,[r4] ; get value of b
ADR r4,c ; get address for c
LDR r1,[r4] ; get value of c
ADD r2,r0,r1 ; compute partial result
ADR r4,a ; get address for a
LDR r0,[r4] ; get value of a
```

C assignment, cont'd.

```
MUL r2,r2,r0 ; compute final value for y
ADR r4,y ; get address for y
STR r2,[r4] ; store y
```

Example: C assignment

- C:

```
z = (a << 2) |  (b & 15);
```

- Assembler:

```
ADR r4,a ; get address for a
LDR r0,[r4] ; get value of a
MOV r0,r0,LSL 2 ; perform shift
ADR r4,b ; get address for b
LDR r1,[r4] ; get value of b
AND r1,r1,#15 ; perform AND
ORR r1,r0,r1 ; perform OR
```

C assignment, cont'd.

```
ADR r4,z ; get address for z
STR r1,[r4] ; store value for z
```

Additional addressing modes

- Base-plus-offset addressing:

```
LDR r0,[r1,#16]
```

  - Loads from location r1+16
- Auto-indexing increments base register:

```
LDR r0,[r1,#16]!
```

- Post-indexing fetches, then does offset:

```
LDR r0,[r1],#16
```

  - Loads r0 from r1, then adds 16 to r1.

Flow of control

- All operations can be performed conditionally, testing CPSR:
  - EQ, NE, CS, CC, MI, PL, VS, VC, HI, LS, GE, LT, GT, LE
- Branch operation:

```
B #100
```

  - Can be performed conditionally.

Example: if statement

- C:

```
if (a > b) { x = 5; y = c + d; } else x = c - d;
```

- Assembler:

```
; compute and test condition
  ADR r4,a ; get address for a
  LDR r0,[r4] ; get value of a
  ADR r4,b ; get address for b
  LDR r1,[r4] ; get value for b
  CMP r0,r1 ; compare a < b
  BGE fblock ; if a >= b, branch to false block
```

If statement, cont'd.

```
; true block
   MOV r0,#5 ; generate value for x
   ADR r4,x ; get address for x
   STR r0,[r4] ; store x
   ADR r4,c ; get address for c
   LDR r0,[r4] ; get value of c
   ADR r4,d ; get address for d
   LDR r1,[r4] ; get value of d
   ADD r0,r0,r1 ; compute y
   ADR r4,y ; get address for y
   STR r0,[r4] ; store y
   B after ; branch around false block
```

If statement, cont'd.

```
; false block
fblock ADR r4,c ; get address for c
  LDR r0,[r4] ; get value of c
  ADR r4,d ; get address for d
  LDR r1,[r4] ; get value for d
  SUB r0,r0,r1 ; compute a-b
  ADR r4,x ; get address for x
  STR r0,[r4] ; store value of x
after ...
```

Example: Conditional instruction implementation

```
; true block
   MOVLT r0,#5 ; generate value for x
   ADRLT r4,x ; get address for x
   STRLT r0,[r4] ; store x
   ADRLT r4,c ; get address for c
   LDRLT r0,[r4] ; get value of c
   ADRLT r4,d ; get address for d
   LDRLT r1,[r4] ; get value of d
   ADDLT r0,r0,r1 ; compute y
   ADRLT r4,y ; get address for y
   STRLT r0,[r4] ; store y
```

Example: switch statement

- C:

```
switch (test) { case 0: … break; case 1: … }
```

- Assembler:

```
   ADR r2,test ; get address for test
   LDR r0,[r2] ; load value for test
   ADR r1,switchtab ; load address for switch table
   LDR r1,[r1,r0,LSL #2] ; index switch table
switchtab DCD case0
   DCD case1
...
```

Example: FIR filter

- C:

```
for (i=0, f=0; i<N; i++)
    f = f + c[i]*x[i];
```

- Assembler

```
; loop initiation code
  MOV r0,#0 ; use r0 for I
  MOV r8,#0 ; use separate index for arrays
  ADR r2,N ; get address for N
  LDR r1,[r2] ; get value of N
  MOV r2,#0 ; use r2 for f
```

FIR filter, cont'.d

```
   ADR r3,c ; load r3 with base of c
   ADR r5,x ; load r5 with base of x
; loop body
loop LDR r4,[r3,r8] ; get c[i]
   LDR r6,[r5,r8] ; get x[i]
   MUL r4,r4,r6 ; compute c[i]*x[i]
   ADD r2,r2,r4 ; add into running sum
   ADD r8,r8,#4 ; add one word offset to array index
   ADD r0,r0,#1 ; add 1 to i
   CMP r0,r1 ; exit?
   BLT loop ; if i < N, continue
```

ARM subroutine linkage

- Branch and link instruction:

```
BL foo
```

  - Copies current PC to r14.

- To return from subroutine:

```
MOV r15,r14
```

Nested subroutine calls

- Nesting/recursion requires coding convention:

```
f1      LDR r0,[r13] ; load arg into r0 from stack
        ; call f2()
        STR r13!,[r14] ; store f1’s return adrs
        STR r13!,[r0] ; store arg to f2 on stack
        BL f2 ; branch and link to f2
        ; return from f1()
        SUB r13,#4 ; pop f2’s arg off stack
        LDR r13!,r15 ; restore register and return
```

Summary

- Load/store architecture
- Most instructions are RISCy, operate in single cycle.
  - Some multi-register operations take longer.
- All instructions can be executed conditionally.