The Microarchitecture Level

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria

Wolfgang.Schreiner@risc.uni-linz.ac.at
http://www.risc.uni-linz.ac.at/people/schreine

Contents

An Example Microarchitecture

The Microarchitecture Level

Implementation of the ISA.

- **Illustration by example:**
  - ISA: integer subset of Java Virtual Machine (IJVM).
  - Microarchitecture: Mic.

- **Microprogram:**
  - Implementation of each IJVM instruction.
  - Controls data path of microarchitecture.

- **Data path:**
  - 32 bit registers (PC, SP, MDR, ...)
  - Drive contents to B bus.
  - Output of ALU drives shifter and then C bus.
 - C bus value can be written to registers.
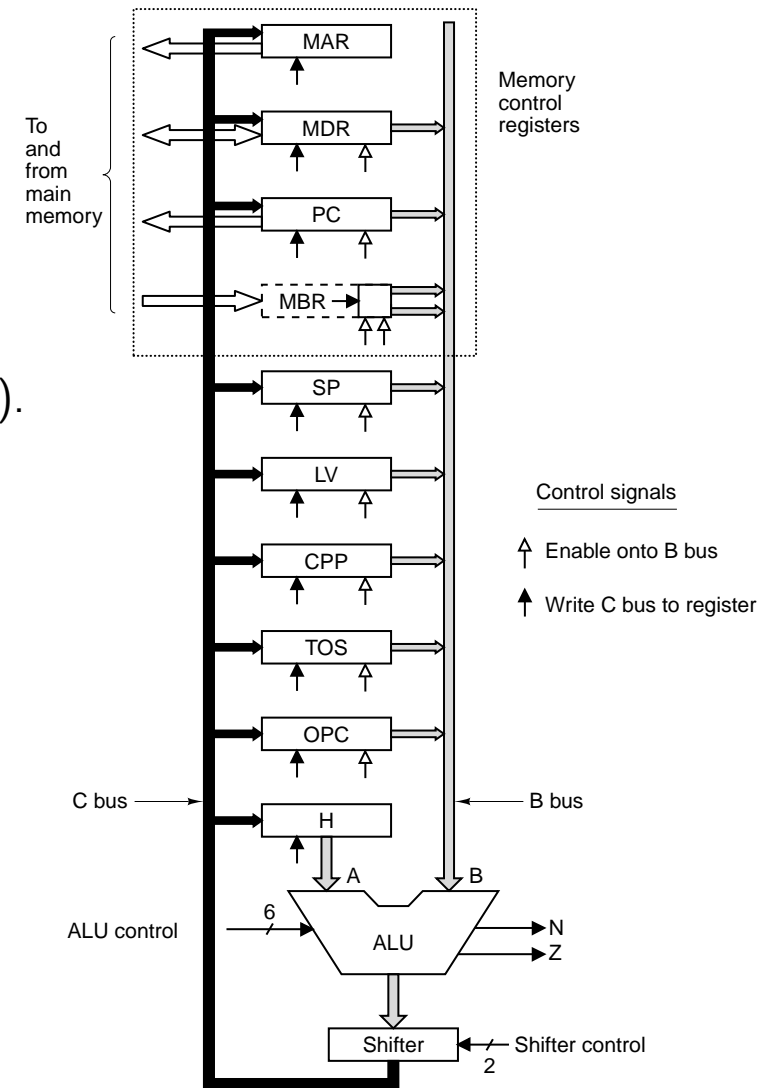
ALU

ALU as presented in the previous section.

- ALU is controlled by six control lines.
  - F0 and F1 determine operation.
  - ENA and ENB enable inputs from bus A respectively bus B.
  - INVA inverts input fromb bus A.
  - INC adds 1 to the result of the operation.

F0	F1	ENA	ENB	INVA	INC	Function
0	1	1	0	1	0	$\bar{A}$
1	1	1	1	0	0	$A + B$
1	1	1	0	0	1	$A + 1$
1	1	1	0	1	1	$-A$
0	0	1	1	0	0	A AND B

ALU can read and write same register in one cycle.

Data Path Timing
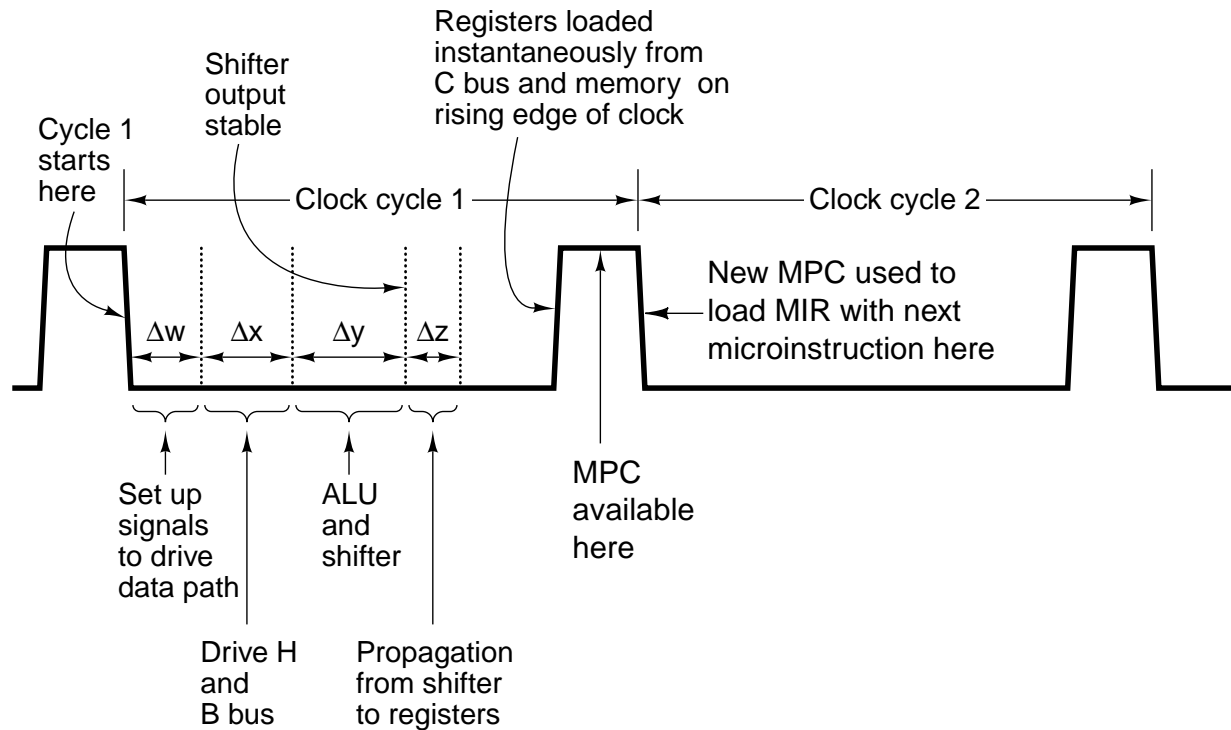
Various phases within one clock cycle.

Data Path Timing

Short pulse is produced at start of each clock cycle.

- Various subcycles:

  1. Control signals are set up ($\triangle w$).
  2. Registers are loaded onto the B bus ($\triangle x$).
  3. The ALU and shifter operate ($\triangle y$).
  4. The results propagate along the C bus to the registers ($\triangle z$).

- Subcycles are implicitly determined by circuit delays:

  – Bits need time to become stable.
  – ALU has some signal propagation time.
    * Until $\triangle w + \triangle x$, ALU input is garbage.
    * Until $\triangle w + \triangle x + \triangle y$, ALU output is garbage.

Operation depends on rigid timing of all elements.
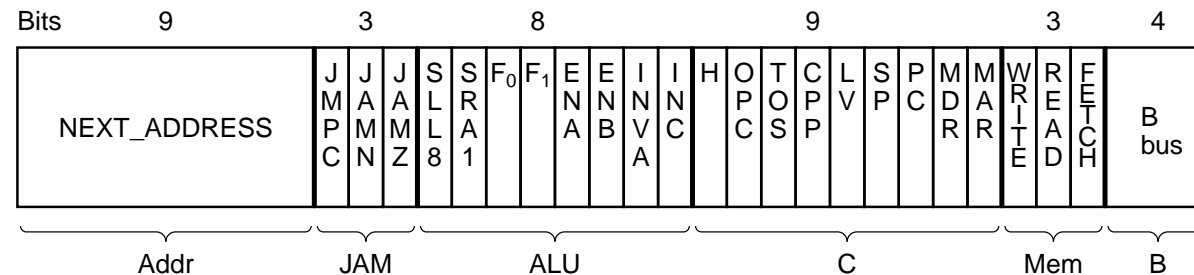
Memory Operation

- 32-bit word-addressable memory port.
  - Controlled by MAR (Memory Address Register) and MDR (Memory Data Register).
  - MAR contains address of word (word 0, word 1, ...) to be read into MDR.
  - Reading and writing of ISA-level data words.
- 8-bit byte-addressable memory port.
  - Controlled by PC (Program Counter).
  - PC contains addresses of byte (byte 0, byte 1, ...) to be read into low-order 8 bits of MBR.
  - Sign extension of MBR (signed, unsigned) is determined by two control lines.
- Each register is driven by one or two control signals.
  - Control signal that enables register's output onto B bus (open arrow).
  - Control signal that loads the register from the C bus (solid arrow).
  - Reading and writing of ISA-level program (byte stream).

Data Path Control

- **29 control signals** determine data path.

  – 9 signals to control writing data from C bus into registers.

  – 9 signals to control enabling registers onto the B bus for ALU input.

  – 8 signals to control ALU and shifter operation.

  – 2 signals to indicate memory read/write via MAR/MDR (not shown).

  – 1 signal to indicate memory fetch via PC/MBR (not shown).

- Signal values specify operations for one cycle of data path.

  – Put values from registers to C bus, propagate signals through ALU and shifter on C bus, write results into appropriate register(s).

- Memory read data signal is asserted in cycle k:

  – Memory operation is started at end of cycle k (after MAR has been loaded).

  – Memory data are available in MDR at the very end of cycle $k + 1$.

  – Memory data can be used in cycle $k + 2$.

Microinstructions

Can reduce number of bits needed for control.

Bits	9	3	8	9	3	4
	NEXT_ADDRESS	J M P C / J A M N / J A M Z	S L L 8 / S R A 1 / F_0 / F_1 / E N A / E N B / I N V A / I N C	H / O P C / T O S / C P P / L V / S P / P C / M D R / M A R	W R I T E / R E A D / F E T C H	B bus
	Addr	JAM	ALU	C	Mem	B

B bus registers

0 = MDR	5 = LV
1 = PC	6 = CPP
2 = MBR	7 = TOS
3 = MBRU	8 = OPC
4 = SP	9-15 none

- Only one of the nine registers can drive B bus.

  – Only four bits are needed to select one of the registers (B).

  – Decoder generates from four bit value one of the 9 output signals.

- Data path can be controlled by 24 signals.

  – First part of a micro-instruction (ALU, C, Mem, B).

  – Second part determines which micro-instruction is executed next (Addr, JAM).

Microinstruction Control

Which control signals should be enabled on each cycle?

- **Sequencer** steps through microinstructions.

  1. Determines state of every control signal.
  2. Determines address of microinstruction to be executed next.

- **Control store** holds complete microprogram.

  - Like program memory, but microinstructions instead of ISA instructions.
  - 512 words containing 36-bit microinstructions.
  - Each microinstruction determines next microinstruction to be executed.

- **MPC (MicroProgram Counter), MIR (MicroInstruction Counter)**

  - MPC: Address of next microinstruction to be fetched from memory.
  - MIR: Current microinstruction whose bits drive control signals of data path.

Microarchitecture

Mic-1.
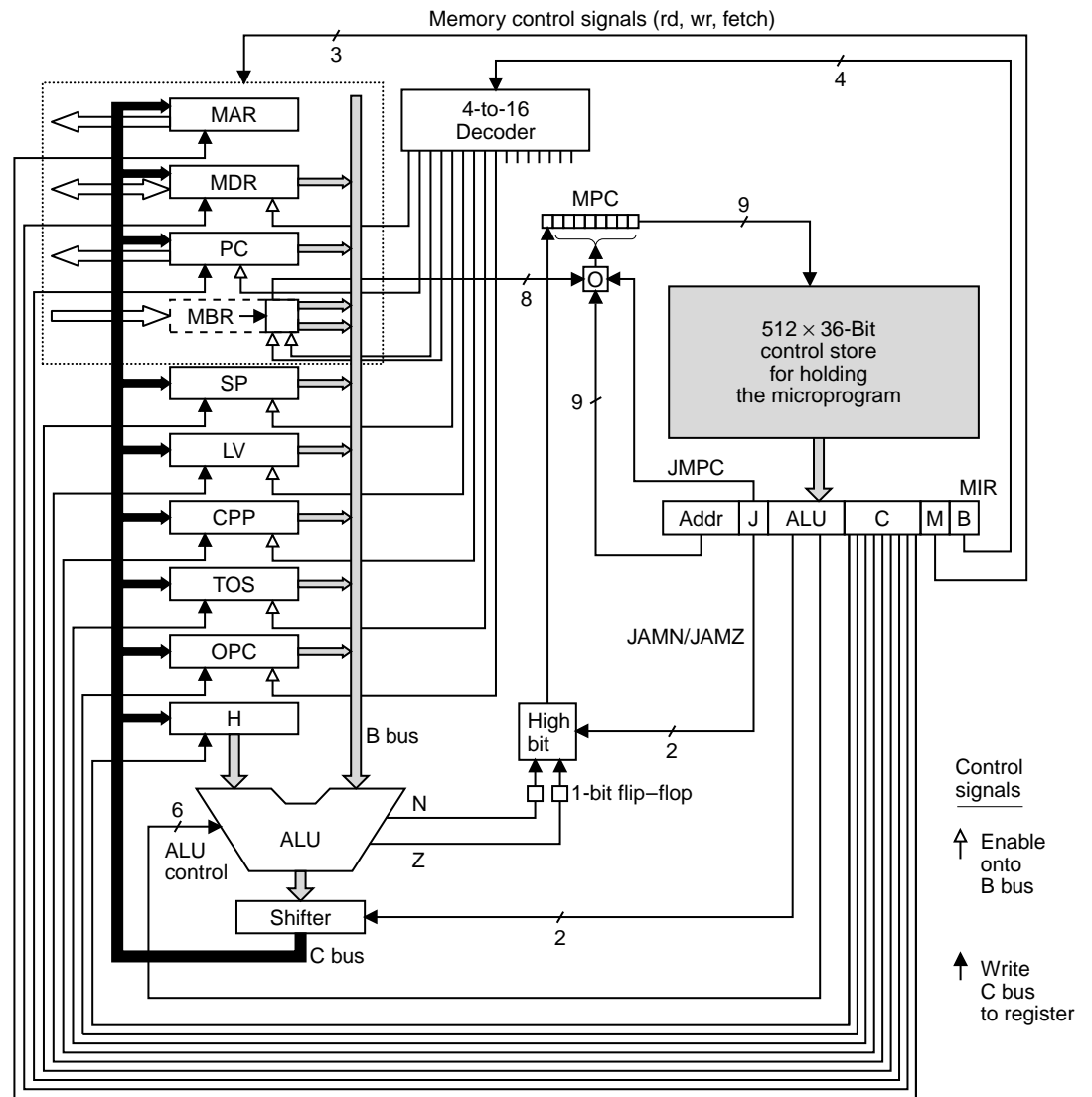
Mic Operation

1. MIR is loaded from the word in control store pointed to by MPC.

   - By $\triangle w$, MIR is loaded.

2. Control signals propagate from MIR into the data path.

   - One register is put onto the B bus.
   - ALU is told which operation to perform.
   - By $\triangle w + \triangle x$, ALU inputs are stable.

3. ALU and shifter execute.

   - By $\triangle w + \triangle x + \triangle y$, ALU and shifter output are stable.

4. Output is written into registers.

   - By $\triangle w + \triangle x + \triangle y + \triangle z$, shifter output has reached registers and N and Z flip-flops.

After subcycle 4, MPC for next microinstruction is determined.
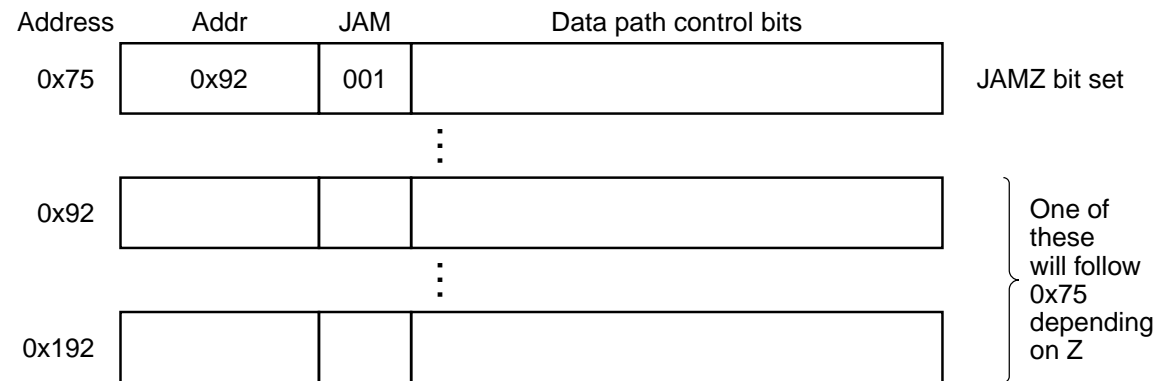
Microinstruction Sequencing

Microinstructions are not implicitly sequenced.

- NEXT_ADDRESS field is copied to MPC.

    – Simultaneously, JAM field is inspected.

- Case: JAM=0.

    – Nothing else is done.

- Case: JAM$\neq$ 0.

    – JAMN=1: 1-bit N flip-flop is ORed into high-order bit of MPC.
    – JAMZ=1: 1-bit Z flip-flop is ORed into high-order bit of MPC.
    – If both JAMN and JAMZ is set, both bits are ORed there.
    – MPC[8] := (JAMZ AND Z) OR (JAMN AND N) OR NEXT_ADDRESS[8].

MPC becomes NEXT_ADDRESS (high-order bit potentially set to 1).

Microinstruction Sequencing

- **If JAMN/JAMZ bit is set, two successor instructions are possible.**
  - JAMN bit is set: successor instruction depends on value of N bit set by ALU.
    * N Bit is set if ALU result is negative.
  - JAMZ bit is set: successor instruction depends on value of Z bit set by ALU.
    * Z Bit is set if ALU result is zero.

Address	Addr	JAM	Data path control bits	
0x75	0x92	001		JAMZ bit set
		⋮		
0x92				One of these will follow 0x75 depending on Z
		⋮		
0x192				

- **If JMPC bit is set, 256 successor instructions are possible.**
  - MPC := NEXT_ADDRESS || 0:MBR

An Example Instruction Set

Example ISA: IJVM

Instruction set to be interpreted by Mic microprogram.

- **Stack:** memory area for local variables.
  - Local variables of methods cannot be stored at absolute addresses.
  - Two invocations of methods may be active at same time (recursion).
  - Local variables are organized in a stack-like fashion.
- **Local variable frame:** variables of "current" procedure activation.
  - Determined by two registers LV and SP.
  - LV points to the base of the local variable frame.
  - SP points to the highest word of the frame.
  - Variables are referred to by their offset from LV.

Programming languages are implemented with the use of stacks.

Variable Stack
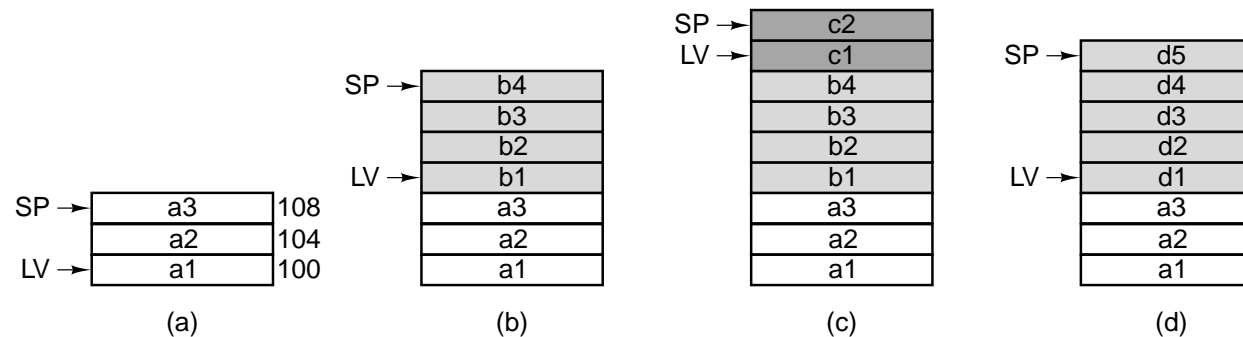
1. Program calls procedure A with three local variables (a_1, a_2, a_3).

2. A calls procedure B with local variables (b_1, b_2, b_3, b_4).

3. B calls procedure C with local variables (c_1, c_2).

4. C and B return; A calls D with local variables $(d_1, d_2, d_3, d_4, d5)$.

Variable frames are pushed on and popped of the stack.

Operand Stack

Stack also holds operands during arithmetic computations.
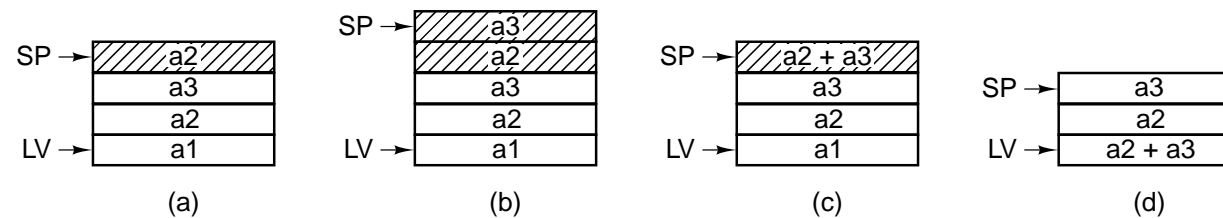
$$a_1 = a_2 + a_3;$$

1. Push a_2 onto the stack.

2. Push a_3 onto the stack.

3. Pop two words off the stack, add them, push result onto the stack.

4. Pop top word off the stack and store it in local variable a_1.

Local variable frames and operand stacks are intermixed.

The IJVM Memory Model

- Constant pool.

    - Read-only memory for constants, strings, etc.
    - Addressed by register CPP.

- Local variable frame.

    - Memory for local variables of current method invocation.
    - Addressed by register LV.

- Operand stack.

    - Allocated on top of local variable frame.
    - Top address denoted by register SP.

- Method area.

    - Memory for program code.
    - Current instruction denoted by register PC.
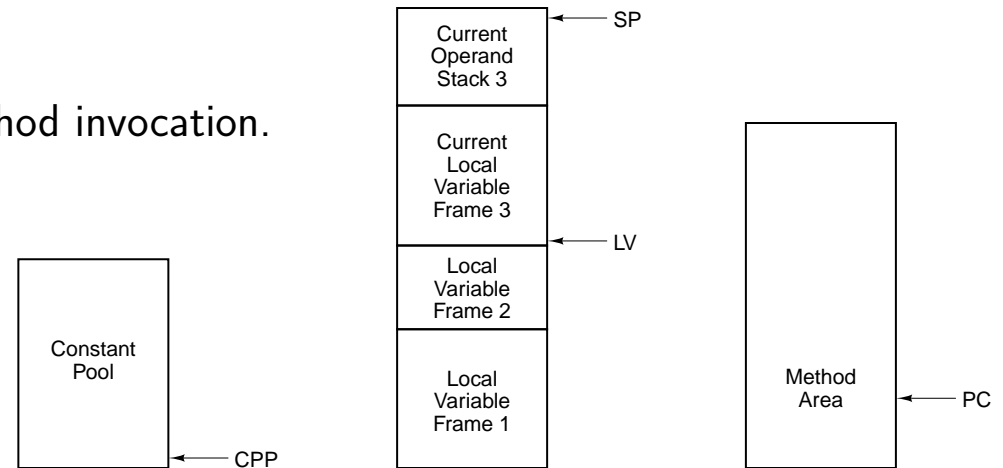
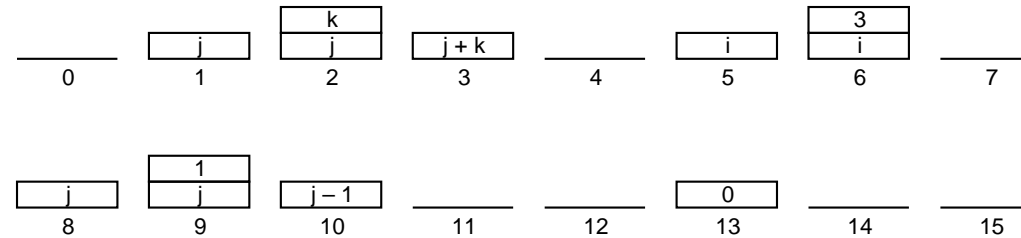CPP, LV, SP address *words*; PC addresses *bytes*.

The IJVM Instruction Set

Hex	Mnemnonic	Meaning
0x10	BIPUSH *byte*	Push byte onto stack
0xA7	GOTO *offset*	Unconditional branch
0x60	IADD	Pop two words from stack; push their sum
0x64	ISUB	Pop two words from stack; push their difference
0x9F	IF_ICMPEQ *offset*	Pop two words from stack; branch if equal
0x15	ILOAD *varnum*	Push local variable onto stack
0x36	ISTORE *varnum*	Pop word from stack and store in local variable
0xB6	INVOKEVIRTUAL *disp*	Invoke a method
0xAC	IRETURN	Return from method with integer value
. . .		

Some instructions have an operand (memory offset, constant).

Example

```
                          ┌─────┐               ┌─────┐
                          │  k  │               │  3  │
       ┌─────┐   ┌─────┐  ├─────┤  ┌─────┐      ┌─────┐  ┌─────┐
 ____  │  j  │   │  j  │  │ j+k │  │  i  │ ____  │  i  │  ____
  0       1         2        3      4      5       6       7
```
```
       ┌─────┐   ┌─────┐
       │  1  │
 ┌─────┐  ├─────┤  ┌─────┐
 │  j  │  │  j  │  │ j−1 │  ____   ____   ┌─────┐  ____   ____
                                          │  0  │
   8       9        10      11     12        13     14     15
```

| Java	IJVM memnonic			IJVM hexadecimal
i = j+k;	1	ILOAD j	// i = j+k	0x15 0x02
if (i == 3)	2	ILOAD k		0x15 0x03
k = 0;	3	IADD		0x60
else	4	ISTORE i		0x36 0x01
j = j−1;	5	ILOAD i	// if (i==3)	0x15 0x01
	6	BIPUSH 3		0x10 0x03
	7	IF_ICMPEQ L1		0x9F 0x00 0x0D
	8	ILOAD j	// j = j−1	0x15 0x02
	9	BIPUSH 1		0x10 0x01
	10	ISUB		0x64
	11	ISTORE j		0x36 0x02
	12	GOTO L2		0xA7 0x00 0x07
	13 L1:	BIPUSH 0	// k = 0	0x10 0x00
	14	ISTORE k		0x36 0x03
	15 L2:			

INVOKEVIRTUAL

- **Method invocation.**

  1. Push pointer to object.
  2. Push method parameters.
  3. Execute INVOKEVIRTUAL.

- **INVOKEVIRTUAL** *disp*:

  - *disp*: position in constant pool.
  - Contains start address of method.
    * Actual method code starts at byte 5.
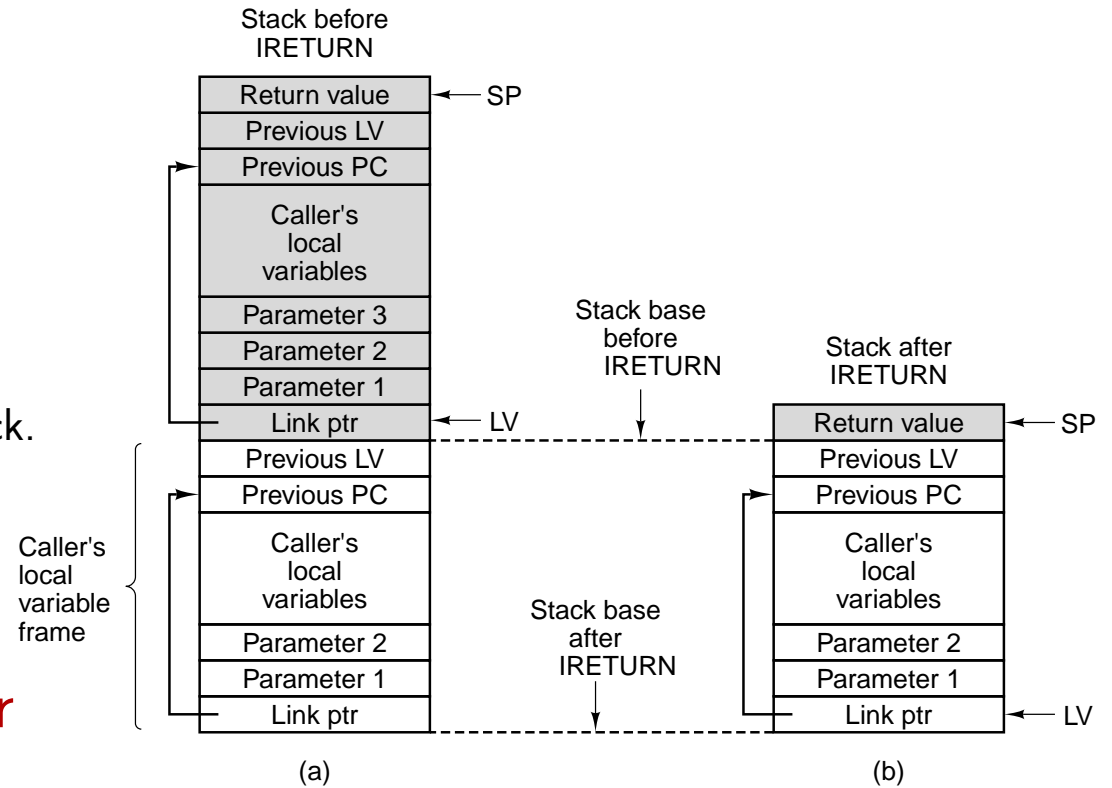   * Bytes 0–1: number of parameters; bytes 2-3: size of local variable frame.
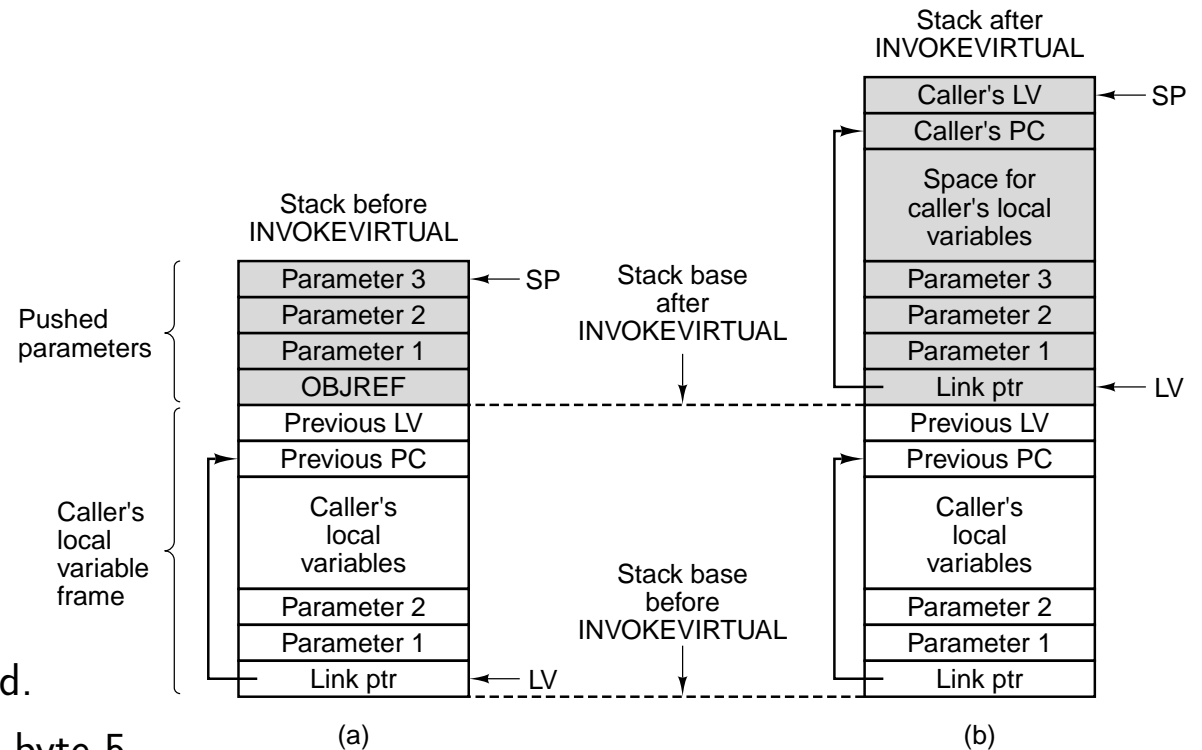
  1. Set SP to new top of stack.
  2. Store old LV and old PC in frame.
  3. Set LV to base of new frame.
  4. Set PC to byte 4 in method code.

Stack after INVOKEVIRTUAL

Caller's LV	← SP
Caller's PC	
Space for caller's local variables	
Parameter 3	
Parameter 2	
Parameter 1	
Link ptr	← LV
Previous LV	
Previous PC	
Caller's local variables	
Parameter 2	
Parameter 1	
Link ptr	

(b)

Stack before INVOKEVIRTUAL

Pushed parameters:
Parameter 3	← SP
Parameter 2	
Parameter 1	
OBJREF	

Stack base after INVOKEVIRTUAL

Caller's local variable frame:
Previous LV	
Previous PC	
Caller's local variables	
Parameter 2	
Parameter 1	
Link ptr	← LV

Stack base before INVOKEVIRTUAL

(a)

IRETURN

- Restores original state.

  1. Deallocates space.
  2. Restores stack to former state.
  3. Places return value on top of stack.
  4. Restores PC.

Execution continues with instruction immediately after INVOKEVIRTUAL.

Stack before IRETURN

Return value	← SP
Previous LV	
Previous PC	
Caller's local variables	
Parameter 3	
Parameter 2	
Parameter 1	
Link ptr	← LV

Stack base before IRETURN

Stack after IRETURN

Caller's local variable frame

Previous LV	
Previous PC	
Caller's local variables	
Parameter 2	
Parameter 1	
Link ptr	

Stack base after IRETURN

Return value	← SP
Previous LV	
Previous PC	
Caller's local variables	
Parameter 2	
Parameter 1	
Link ptr	← LV

(a)   (b)

Implementation of the Instruction Set

The MIC Microprogram

Use symbolic notation for micro-instructions.

- Text line specifies all activites that occur in single clock cycle.
  - We want to increment the value of SP, to initiate a read operation, and to branch to the microinstruction at location 122 in the control store.

- List the signals that are activated in clock cycle:

  ```
  ReadRegister = SP, ALU = INC, WSP, Read, NEXT_ADDRESS = 122
  ```

- We will use a shorter version:

  ```
  SP = SP+1; rd
  ```

Micro assembly language MAL for writing micro-programs.

The Mic Registers

- ## CPP, LV, and SP.

  – Point to constant pool, local variable frame, and top of stack.

- ## PC (program counter)

  – Holds address of next byte to be fetched from instruction stream.

- ## MBR (memory byte register)

  – One byte register that holds the bytes of the instruction stream as they are interpreted.

- ## TOS (top of stack)

  – Holds at beginning/end of each instruction value of the memory location pointed to by SP.

- ## OPC

  – Temporary (scratch) register used, e.g., to save address of opcode for a branch instruction.

The Main Interpreter

Label	Operations	Comments
Main	PC = PC+1; fetch; goto (MBR)	MBR holds opcode; get next byte; dispatch

- Increment the PC to point to the next opcode.
- Initiate a fetch of the next opcode into MBR.
  - Will be needed as an operand or as the next opcode.
- Perform a multiway branch to the address contained in MBR.
  - This is the value placed there by the previous microinstruction.

Micro-code for each IJVM instruction will return to Main.

BIPUSH and GOTO

BIPUSH (0×10)	BYTE

Label	Operations	Comments
BIPUSH	SP = MAR = SP+1	MBR = the byte to push onto stack
	PC = PC+1; fetch	Increment PC; fetch the next opcode
	MDR = TOS = MBR; wr; goto Main	Sign-extend constant and push on stack
GOTO	OPC = PC−1	Save address of opcode
	PC = PC+1; fetch	MBR = 1st byte of offset; fetch 2nd byte
	H = MBR << 8	Shift and save signed first byte in H
	H = MBRU OR H	H = 16-bit branch offset
	PC = OPC+H; fetch	Add offset to OPC
	goto Main	Wait for fetch of next opcode

IADD and ISUB

Label	Operations	Comments
IADD	MAR = SP = SP−1; rd	Read in next-to-top word on stack
	H = TOS	H = top of stack
	MDR = TOS = MDR+H; wr; goto Main1	Add top of two words; write to top of stack
ISUB	MAR = SP = SP−1; rd	Read in next-to-top word on stack
	H = TOS	H = top of stack
	MDR = TOS = MDR−H; wr; goto Main1	Add top of two words; write to top of stack

IF_ICMPEQ

Label	Operations	Comments
IF_ICMPEQ	MAR = SP = SP−1; rd	Read in next-to-top word of stack
	MAR = SP = SP−1	Set MAR to read in new top-of-stack
	H = MDR; rd	Copy second stack word to H
	OPC = TOS	Save TOS in OPC temporarily
	TOS = MDR	Put new top of stack in TOS
	Z = OPC−H; if (Z) goto T; else goto F	If top 2 words are equal,
		goto T, else goto F

Improving Performance

Improving Performance

Various basic techniques.

1. Cache memory.

2. Branch predication.

3. Out-of-order execution with register renaming.

4. Speculative execution.

The first three techniques do not change the architecture (old programs will still run, i.e., programs do not see a difference).

Cache Memory

Cache: small fast memory that holds the most recently used words.

- **Split cache:** multiple caches with independent access to memory.
  - Instruction cache: program words.
  - Data cache: data words.
- **Level 2 cache:** cache between level 1 cache and memory.
  - Memory accesses go first to L1 cache, then to L2 cache, then to memory.
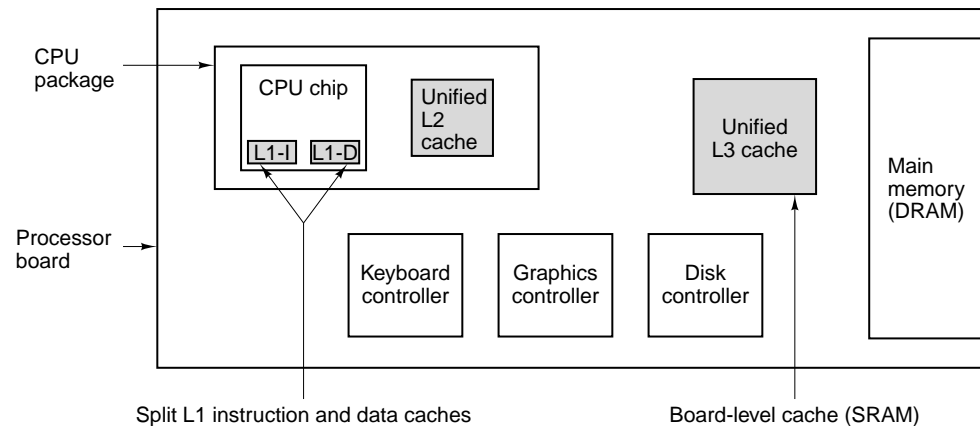
Locality Principles

Caches depend on locality to achieve their goal.

- Spatial locality:
  - Memory locations with addresses numerically similar to recently accessed memory locations are likely to be accessed in the near future.
  - Effect on transfer policy: more data than have been requested are transferred from memory.
- Temporal locality:
  - Recently accessed memory locations are accessed again (memory locations near the top of a stack or instructions inside a loop).
  - Effect on cache miss strategy: not recently accessed cache entries are discarded.

Considering these principles in program development may drastically improve the performance of programs.

Cache Principles

Basic principles of all caches.

- Cache line: fixed size (e.g. 32 bytes) memory block.
  - Main memory is divided into lines: bytes 0–31, 32–63, . . .
  - Some lines are in cache at any time.
- Memory word is referenced by program:
  - Cache controller circuit checks whether the referenced word is in cache.
  - If word is not there, some cache line is removed from cache.
  - Needed line is fetched from memory (or lower level cache).
- Goal: keep most heavily-used lines in cache as much as possible.
  - Maximize number of memory references satisfied out of cache.

Several variations of this scheme.

Direct-Mapped Caches
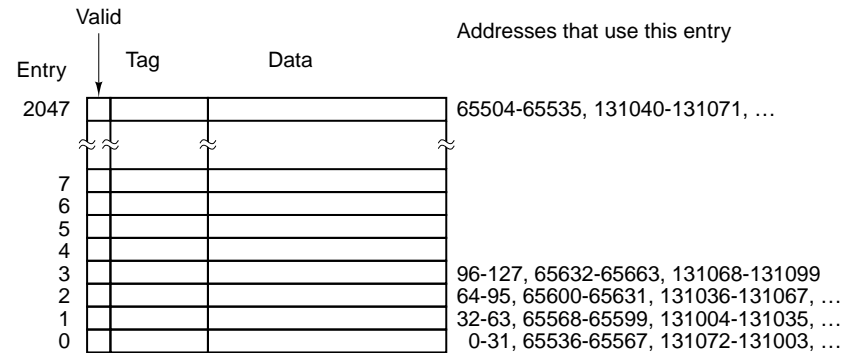
Simplest form of caches.

- Each cache entry consists of:
  - A Valid bit to indicate whether the entry has valid data.
  - A Tag field consisting of a unique value identifiying the memory line.
  - A Data field holding a cache line of 32 bytes.
- A given memory word can be stored in one place of the cache only.
  - The LINE field of its address determines the index of the entry in the cache.
  - The TAG field of its address can be used to check whether the right line is in that entry.
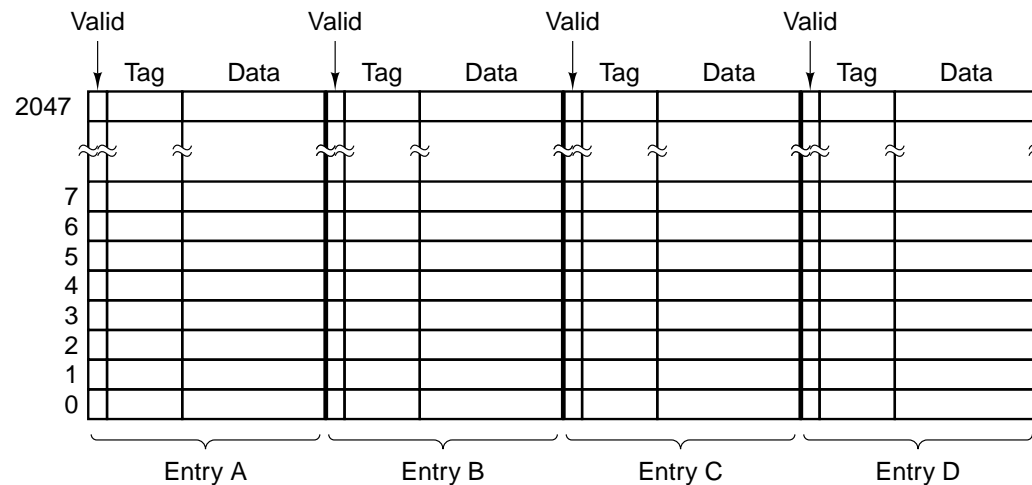 - The word field determines the position in the cache line.
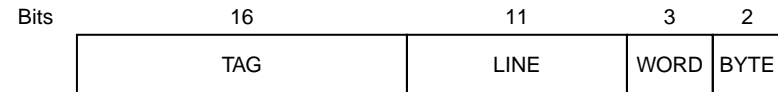
Set-Associative Caches

A word can be stored in a fixed number of places in the cache.

- n-way set-associative cache.
  - There exist n possible entries for a cache line (usually $n = 2$ or $n = 4$).
- LRU (Least Recently Used) Algorithm.
  - If a new cache entry is loaded, it replaces the least recently used of the n candidates.

Branch Prediction

Modern computers are highly pipelined.

- Data path is decomposed.
  - Mic-2: Instruction Fetch Unit (IFU).
  - Mic-3: Registers A, B, C.
- Pipelined design.
  - Parts can be used simultaneously.
  - Clock can be speed up.
    * Maximum delay becomes shorter.
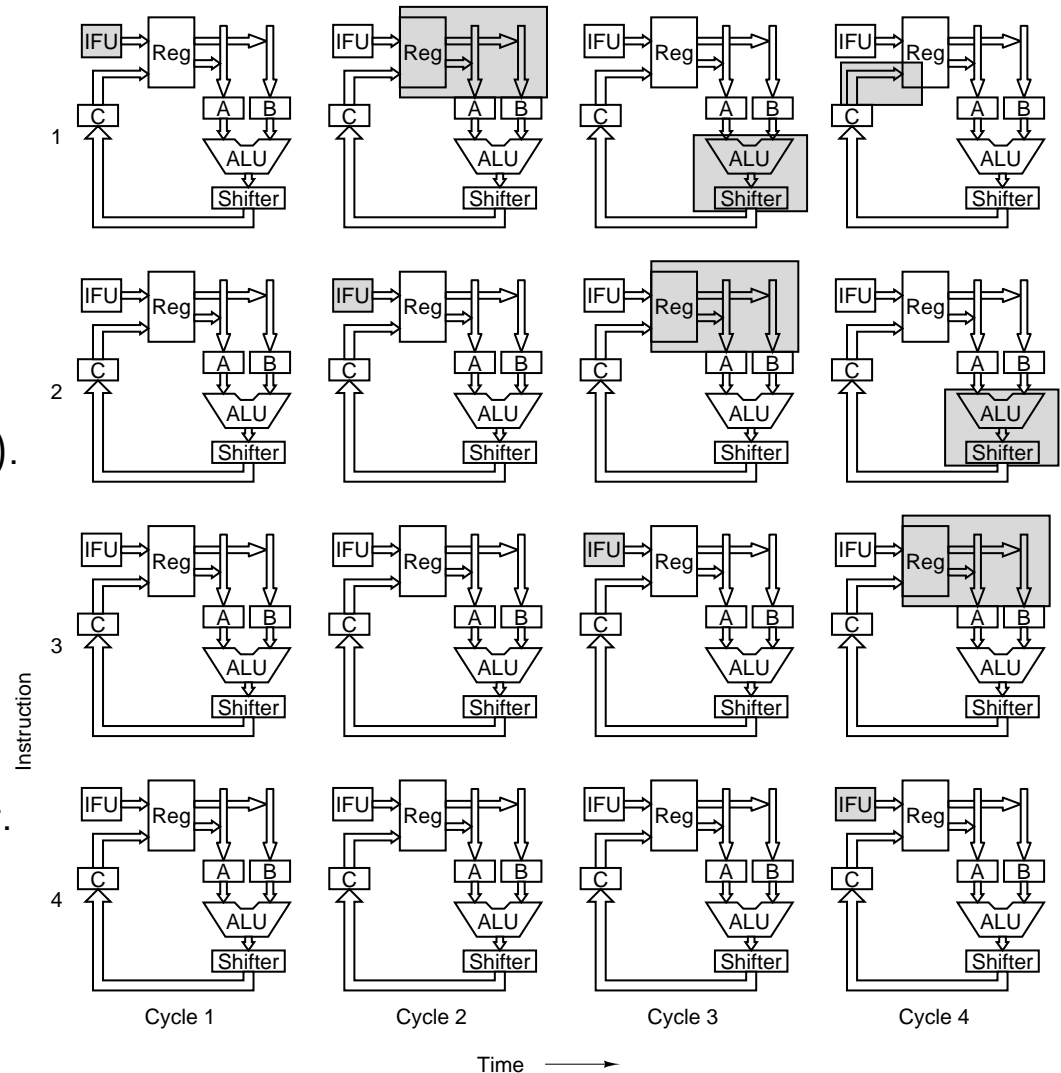
For good performance, keep pipeline full.

Branch Prediction
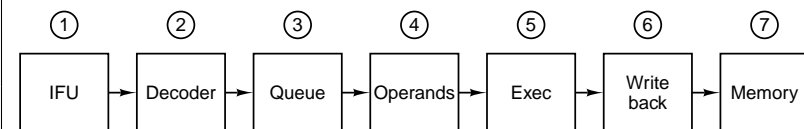
Actual code is full of branch instructions.

```
if (i == 0)              CMP i,0      ; compare i to 0
    k = 1;               BNE Else     ; branch to Else if not equal
else            Then:    MOV k,1      ; move 1 to k
    k = 2;               BR Next      ; branch to Next
                Else:    MOV k,2      ; move 2 to k
                Next:
```

Mic-4: seven stage pipeline

① ② ③ ④ ⑤ ⑥ ⑦

| IFU | → | Decoder | → | Queue | → | Operands | → | Exec | → | Write back | → | Memory |

- **Problem with unconditional branches (BR)**
  - Instruction decoding occurs in second stage when first stage already loads next instruction.
 - Delay slot: instruction after branch is still executed (code has to be appropriately changed).
- **Problem with conditional branches (BNE)**
  - Fetch unit does not know which instruction to load until much later in pipeline.
  - Pipeline has to stall until it is known whether branch is taken or not.

Branch Prediction

Dynamic branch prediction: CPU guesses whether branch is taken.

- Cache-like organized history table records conditional branches.

  1. 1 bit: bit says whether branch was taken last time or not.

  2. 2 bits: second bit says whether guess was true last time.
     - Only if second bit is 0, first bit is changed.
     - Consequence: first error (e.g. at end of loop) does not change prediction.

  3. Prediction bits plus target address of last branch.
     - For branch instructions with computed target addresses.

(a)   (b)   (c)
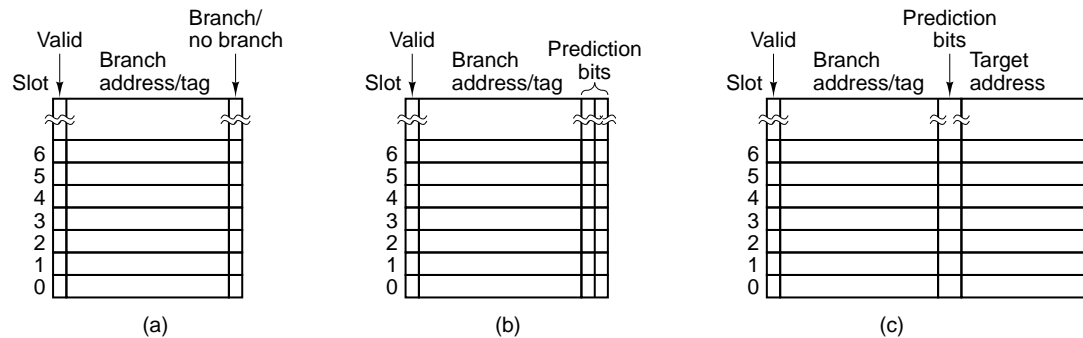
Out-of-Order Execution and Register Renaming

Modern CPUs are super-scalar.

- Multiple functional units are simultaneously fed by decode unit.
  - In-order-execution: instructions are issued in the order they appear in program (1-2-3-4).
  - Instruction 2 may depend on result of instruction 1 and has to wait.
  - Instruction 3 may not depend on instruction 1 but now has to wait as well.
- Out-of-order execution:
  - Decode unit may reorder instructions for feeding functional units (1-3-2-4).
  - Problem: same register R may be written by 1 and 3.
- Register renaming: CPU has secret registers invisible to programs.
  - Decode unit may change instructions to write to/read from secret registers.
  - Instruction 3 is changed to write to S.
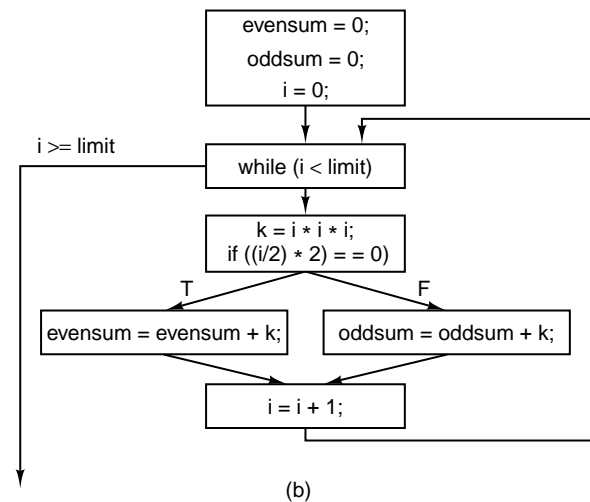 - Instruction 4 reading from R has to be changed to read from S.

Speculative Execution

Out-of-order execution only operates within basic blocks.

```
evensum = 0;

oddsum = 0;

i = 0;

while (i < limit) {

    k = i * i * i;

    if ((i/2) * 2) == 0)

        evensum = evensum + k;

    else

        oddsum = oddsum + k;

    i = i + 1;

}
              (a)
```

```
            evensum = 0;
            oddsum = 0;
               i = 0;

i >= limit
            while (i < limit)

            k = i * i * i;
           if ((i/2) * 2) = = 0)
         T                      F
  evensum = evensum + k;   oddsum = oddsum + k;

               i = i + 1;

                  (b)
```

- **Speculative execution:** code is moved beyond block boundaries.
  - Code may be executed before it is known to get needed.
  - Both branches of conditional may execute simultaneously with block that computes condition.

Requires additional instructions and compiler support.