```python
import torch
import torch.nn as nn

class ResConvBlock(nn.Module):
    '''
    Basic residual convolutional block
    '''
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, 3, 1, 1),
            nn.BatchNorm2d(out_channels),
            nn.GELU(),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(out_channels, out_channels, 3, 1, 1),
            nn.BatchNorm2d(out_channels),
            nn.GELU(),
        )

    def forward(self, x):
        x1 = self.conv1(x)
        x2 = self.conv2(x1)
        if self.in_channels == self.out_channels:
            out = x + x2
        else:
            out = x1 + x2
        return out / torch.sqrt(torch.tensor(2.0, device=out.device))


class UnetDown(nn.Module):
    '''
    UNet down block (encoding)
    '''
    def __init__(self, in_channels, out_channels):
        super(UnetDown, self).__init__()
        layers = [ResConvBlock(in_channels, out_channels), nn.MaxPool2d(2)]
        self.model = nn.Sequential(*layers)

    def forward(self, x):
        return self.model(x)


class UnetUp(nn.Module):
    '''
    UNet up block (decoding)
    '''
    def __init__(self, in_channels, out_channels):
        super(UnetUp, self).__init__()
        layers = [
            nn.ConvTranspose2d(in_channels, out_channels, 2, 2),
            ResConvBlock(out_channels, out_channels),
            ResConvBlock(out_channels, out_channels),
        ]
        self.model = nn.Sequential(*layers)

    def forward(self, x, skip):
        x = torch.cat((x, skip), 1)
        x = self.model(x)
        return x


class EmbedBlock(nn.Module):
    '''
    Embedding block to embed time step/condition to embedding space
```

```python
    '''
    def __init__(self, input_dim, emb_dim):
        super(EmbedBlock, self).__init__()
        self.input_dim = input_dim
        layers = [
            nn.Linear(input_dim, emb_dim),
            nn.GELU(),
            nn.Linear(emb_dim, emb_dim),
        ]
        self.layers = nn.Sequential(*layers)

    def forward(self, x):
        # set embedblock untrainable
        for param in self.layers.parameters():
            param.requires_grad = False
        x = x.view(-1, self.input_dim)
        return self.layers(x)

class FusionBlock(nn.Module):
    '''
    Concatenation and fusion block for adding embeddings
    '''
    def __init__(self, in_channels, out_channels):
        super(FusionBlock, self).__init__()
        self.layers = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, 1),
            nn.BatchNorm2d(out_channels),
            nn.GELU(),
        )
    def forward(self, x, t, c):
        h,w = x.shape[-2:]
        return self.layers(torch.cat([x, t.repeat(1,1,h,w), c.repeat(1,1,h,w)], dim = 1))

class ConditionalUnet(nn.Module):
    def __init__(self, in_channels, n_feat = 128, n_classes = 10):
        super(ConditionalUnet, self).__init__()

        self.in_channels = in_channels
        self.n_feat = n_feat
        self.n_classes = n_classes

        # embeddings
        self.timeembed1 = EmbedBlock(1, 2*n_feat)
        self.timeembed2 = EmbedBlock(1, 1*n_feat)
        self.conditionembed1 = EmbedBlock(n_classes, 2*n_feat)
        self.conditionembed2 = EmbedBlock(n_classes, 1*n_feat)

        # down path for encoding
        self.init_conv = ResConvBlock(in_channels, n_feat)
        self.downblock1 = UnetDown(n_feat, n_feat)
        self.downblock2 = UnetDown(n_feat, 2 * n_feat)
        self.to_vec = nn.Sequential(nn.AvgPool2d(7), nn.GELU())


        # up path for decoding
        self.upblock0 = nn.Sequential(
            nn.ConvTranspose2d(2 * n_feat, 2 * n_feat, 7, 7),
            nn.GroupNorm(8, 2 * n_feat),
            nn.ReLU(),
        )
        self.upblock1 = UnetUp(4 * n_feat, n_feat)
        self.upblock2 = UnetUp(2 * n_feat, n_feat)
        self.outblock = nn.Sequential(
            nn.Conv2d(2 * n_feat, n_feat, 3, 1, 1),
            nn.GroupNorm(8, n_feat),
            nn.ReLU(),
            nn.Conv2d(n_feat, self.in_channels, 3, 1, 1),
```

```python
        )

        # fusion blocks
        self.fusion1 = FusionBlock(3 * self.n_feat, self.n_feat)
        self.fusion2 = FusionBlock(6 * self.n_feat, 2 * self.n_feat)
        self.fusion3 = FusionBlock(3 * self.n_feat, self.n_feat)
        self.fusion4 = FusionBlock(3 * self.n_feat, self.n_feat)

    def forward(self, x, t, c):
        '''
        Inputs:
            x: input images, with size (B,1,28,28)
            t: input time stepss, with size (B,1,1,1)
            c: input conditions (one-hot encoded labels), with size (B,10)
        '''
        t, c = t.float(), c.float()

        # time step embedding
        temb1 = self.timeembed1(t).view(-1, self.n_feat * 2, 1, 1)
        temb2 = self.timeembed2(t).view(-1, self.n_feat, 1, 1)

        # condition embedding
        cemb1 = self.conditionembed1(c).view(-1, self.n_feat * 2, 1, 1)
        cemb2 = self.conditionembed2(c).view(-1, self.n_feat, 1, 1)

        # ==================================================== #
        # YOUR CODE HERE:
        #   Define the process of computing the output of a
        #   this network given the input x, t, and c.
        #   The input x, t, c indicate the input image, time step
        #   and the condition respectively.
        # A potential format is shown below, feel free to use your own ways to design it.
        # down0 =
        # down1 =
        # down2 =
        # up0 =
        # up1 =
        # up2 =
        # out = self.outblock(torch.cat((up2, down0), dim = 1))
        # ==================================================== #

        # Encoder
        down0 = self.init_conv(x)                           # (B, 128, 28, 28)
        down1 = self.downblock1(down0)                      # (B, 128, 14, 14)
        down1 = self.fusion1(down1, temb2, cemb2)           # (B, 128, 14, 14)
        down2 = self.downblock2(down1)                      # (B, 256, 7, 7)
        down2 = self.fusion2(down2, temb1, cemb1)           # (B, 256, 7, 7)

        # Bottleneck
        hidden = self.to_vec(down2)                         # (B, 256, 1, 1)
        up0 = self.upblock0(hidden)                         # (B, 256, 7, 7)

        # Decoder
        up1 = self.upblock1(up0, down2)                     # (B, 128, 14, 14)
        up1 = self.fusion3(up1, temb2, cemb2)               # (B, 128, 14, 14)
        up2 = self.upblock2(up1, down1)                     # (B, 128, 28, 28)
        up2 = self.fusion4(up2, temb2, cemb2)               # (B, 128, 28, 28)

        # Output
        out = self.outblock(torch.cat((up2, down0), dim=1))  # (B, 1, 28, 28)

        return out
```